НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет електроніки
Кафедра акустичних та мультимедійних електронних систем
(повна назва кафедри)

«До захисту допущено»
Завідувач кафедри
_____ Найда С.А
(підпис) (ініціали, прізвище)

«01» червня 2020 р.

Дипломна робота

на здобуття ступеня бакалавра

зі спеціальності 171 Електроніка (Електронні та інформаційні системи і технології телебачення, кінематографії та звукотехніки) _____

на тему: «Аналіз технології створення базового програмного забезпечення для комп'ютерних ігор» _____

Виконав: студент IV курсу, групи ДВ-61
(шифр групи)

Купленко Д.А. _____
(прізвище, ім'я, по батькові) (підпис)

Керівник дтн. проф. Власюк Г.Г. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант З старший викладач Батіна О.А. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент доцент каф. ЕПС, к.т.н., доц. Катерина КЛЕН
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____ (Купленко Д.А.)
(підпис)

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут (факультет) Факультет електроніки
(повна назва)

Кафедра Кафедра акустичних та мультимедійних електронних систем
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 171 Електроніка (Електронні та інформаційні системи і технології телебачення, кінематографії та звукотехніки)
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Найда С.А
(підпис) (ініціали, прізвище)
« 25 » травня 2020 р.

ЗАВДАННЯ

на дипломну роботу студенту

Купленко Дмитру Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Аналіз технології створення базового програмного забезпечення для комп'ютерних ігор»

керівник роботи: Власюк Ганна Григорівна, дтн. проф. _____ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «25» травня 2020р. №1196-с

2. Термін подання студентом роботи " 01" червня 2020 року

3. Вихідні дані до роботи : Java, OpenGL, LWJGL, Shading language, Intelij IDEA.

4. Зміст роботи: Етапи розвитку ігрової індустрії, перелік ігрових жанрів, розвиток ігрових рушіїв, основні складові елементи ігрових рушіїв, розробка власного графічного ігрового рушія для створення RPG ігор.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) . Презентація на 10 слайдах.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 11 березня 2020р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Написання першого розділу: “1. АНАЛІТИЧНИЙ ОГЛЯД ІГРОВОЇ ІНДУСТРІЇ ”	20.04.2020	Виконано
2	Написання другого розділу: “2 ТЕХНІЧНІ АСПЕКТИ РЕАЛІЗАЦІЇ ІРОВИХ РУШІЇВ ”	15.05.2020	Виконано
3	Написання третього розділу: “3 РОЗРОБКА ГРАФІЧНОГО РУШІЯ ”	31.05.2020	Виконано
4	Підготовка матеріалів до друку та оформлення пояснювальної записки	01.06.2020	Виконано
5	Підготовка доповіді до захисту та оформлення плакатів	02.06.2020	Виконано

Студент _____ Купленко Д.А.

(підпис) (ініціали, прізвище)

Керівник роботи _____ Власюк Г.Г.

(підпис) (ініціали, прізвище)

УДК 004.921

РЕФЕРАТ

Дипломна робота: 123 с., 29 рис., 2 дод., 12 джерел.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ЗАСОБИ РОЗРОБКИ ІГОР, КОМПЮТЕРНІ ІГРИ, ГРАФІЧНИЙ РУШІЙ, ГРАФІЧНА БІБЛІОТЕКА, ІГРОВИЙ РУШІЙ, JAVA, LWJGL, OPENGL, ІГРОВА СЦЕНА.

Об'єкт дослідження – ігрові рушії та засоби розробки графічних ігрових рушіїв.

Метою роботи є дослідження засобів розробки базового програмного забезпечення для комп’ютерних ігор на основі аналізу жанрової класифікації ігор, і розробка власного графічного рушія з використанням пакету бібліотек lwjgl.

Методом дослідження є аналіз літератури, досліджень і розробок відомих компаній на предмет пошуку найбільш ефективних засобів розробки базового програмного забезпечення для комп’ютерних ігор.

В результаті виконання дипломної роботи був розроблений графічний ігровий рушій, який спеціалізується на RPG-дослідницьких іграх з великою ігровою картою, який з невеликим доопрацюванням можна перетворити в комплексний ігровий рушій, для створення складних ігрових сцен, та використовувати для роботи у ААА проектах. Також було запропоновано комп’ютерний практикуми з використання доопрацювання донного рушія для вивчення дисципліни «Технології створення освітніх комп'ютерних ігор та проектування доповненої реальності».

ABSTRACT

The object of research - game engines and tools for developing graphic game engines.

The aim of the work is to study the means of developing basic software for computer games based on the analysis of genre classification of games, and to develop your own graphical engine using the lwjgl library package.

The research method is the analysis of the literature, research and development of well-known companies in order to find the most effective means of developing basic software for computer games.

As a result of the thesis, a graphic game engine was developed, which specializes in RPG-research games with an open world, which with a little refinement can be turned into a complex game engine to create complex game scenes and use to work in AAA projects. A computer workshop on the use of bottom engine refinement to study the discipline "Technologies for creating educational computer games and designing augmented reality" was also proposed.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

BSP	-	Binary space partitioning
FMV	-	Full-motion video
FPS	-	First-person shooter
FSAA	-	Fast approXimate Anti-Aliasing
GLSL	-	OpenGL Shading Language
GUI	-	Game user interface
HDR	-	High Dynamic Range
HUD	-	heads-up display
IGC	-	In-game cinematics
LOD	-	level of detail
LWJGL	-	Lightweight Java Game Library
MMOFPS	-	Massively multiplayer online first-person shooter
MMOG	-	Massively multiplayer online game
MMORPG	-	Massively multiplayer online role-playing game
MMORTS	-	Massively multiplayer online real-time strategy
ODE	-	Open Dynamics Engine
PC	-	Personal computer
PVS	-	potentially visible set
RPG	-	Role play game
RTS	-	Real-time strategy
SDK	-	Software development kits
STL	-	Standard template library
VAO	-	Vertex Arrays Object
VBO	-	Vertex Buffer Object
VGA	-	Video Graphics Array
VOIP	-	Voice over IP
VR	-	Virtual reality

ЕОМ	-	Електрична обчислювальна машина
ЕПТ	-	Електро-променева трубка
ПК	-	Персональний комп’тер

ВСТУП

На сьогодні, майже кожна людина грала в комп’ютерні ігри, а термін чула кожна. Ігри сьогодні мають окрему ланку в індустрії економіки, адже прибуток, що отримує видавець гри може окупати бюджет грив в десятки разів. А бюджети сучасних ААА або високо бюджетних ігор складає дев’яти значну суму, і поступово починає обходити кінематограф. Адже ігри на відміну від фільмів розраховані на велику кількість ігрових часів, і окрім рухомої картинки, надають можливість користувачам взаємодії з ними, крім того ігри надають доступ гри в них в будь-який момент часу.

Однак щоб створити гру, необхідний ігровий рушій – базове програмне забезпечення комп'ютерної гри. Термін «Ігровий рушій» виник у середині 1990-х у відношенні до шутерів від першої особи (FPS). Коли гравцям надали можливість змінювати дизайн рівнів, вигляд головного персонажу, та ворожих монстрів. Так виникла спільноти розробників модифікацій — групи окремих людей та невеликих незалежних студій, які будували нові ігри, модифікуючи наявні ігри, використовуючи безкоштовні набори інструментів, надані оригінальними розробниками.

Проте такі модифікації мали рамки, і змінювали лише візуальну складову гри. Тому деякі компанії почали створювати ігрові рушії для можливості створення ігор будь-яких жанрів. Недоліком таких рушіїв є те що вони використовують набагато більше ресурсів, для проектів, ніж вузько направлені рушії, та мають певні обмеження що закладені в основі коду.

Тому великі компанії розроблюють власні ігрові рушії під кожний жанр своїх ігор. І це є відмінним варіантом для створення комплексних, високо бюджетних ігор, що не мають обмежень в ігровій механіці.

Провідні компанії надають перевагу власним або вузькоспеціалізованим рушіям. Оскільки для створення комплексної високо бюджетної гри, необхідно щоб рушій не мав обмежень для механіки гри, і в той час гра могла запускатись на більшості комп'ютерів. І розробка власного рушія дасть змогу створити таку гру,

не використовуючи загально спрямованих рушіїв. Окрім цього на основі сласного ігрового рушія можна створити свою ігрову фірму, та надавати рушій у використання іншими фірмами.

Метою дослідження є аналіз та застосування сучасних графічних бібліотек для створення графічного рушія, що дасть змогу створювати ігри що будуть базуватись на жанрі RPG і його механіках.

Для досягнення поставленої мети були сформульовані такі завдання:

1. дослідити сучасні комп’ютерні ігри;
2. визначити жанри комп’ютерних ігор;
3. проаналізувати складові ігрових рушіїв;
4. дослідити технології для створення ігрових рушіїв;
5. Провести аналіз готових рішень для ігрових рушіїв;

1 АНАЛІТИЧНИЙ ОГЛЯД ІГРОВОЇ ІНДУСТРІЇ

1.1 Визначення поняття комп’ютерної гри

Комп’ютерна гра — це електронна гра, яка передбачає взаємодію з інтерфейсом користувача для отримання візуального зворотного зв’язку на дво- або тривимірному пристрої зображення відео, таких як сенсорний екран, гарнітура віртуальної реальності або монітор чи телевізор.[2]

В наші дні величезна кількість різних за інтересами людей часто грає в комп'ютерні. Серед гравців зустрічаються і бізнесмени, і політики, і домогосподарки, і інженери, і художники — в цілому абсолютно різні люди. Комп'ютерні ігри стали справжнім культурним феноменом — виникнувши як нехитрий плід творчої думки програмістів, вони з кожним роком набували все більшої популярності — і розвинулися до того, що стали окремою специфічною спортивною дисципліною — кіберспорт. Попит народжує пропозицію — і ось по всьому світу зросли компанії з розробки ігор, а робота гейм-дева (gamedevelopment) стала відмінною перспективою для багатьох юних умів, які бажають створювати комп'ютерні іграшки. Деякі ігрові серії стали культовими — наприклад, DOOM, Quake, Civilization, HoMM, Fallout, Metal Gear, Dragon Quest, Legend of Zelda, Final Fantasy, TES, CoD, Half-Life, CS, WoW, Starcraft, Diablo, NFS, GTA. Як мінімум про одну з них напевно чула будь-яка людина, яка хоч раз стикалась з комп'ютером.

1.2 Історія розвитку ігрової індустрії

Історія розвитку відеоігор налічує вже понад 60 років. Перші ігри були примітивними, проте саме вони задали вектор розвитку всієї ігрової індустрії.

Еволюція відеоігор безпосередньо пов'язана з прогресом у сфері «ігрового заліза». Причому в початковій стадії це стосувалося, в першу чергу, ігрових приставок, тому що ПК стали доступні більшості користувачів набагато пізніше.

Створення перших відеоігор передували деякі напрацювання і винаходи, завдяки яким і з'явилися на світ «прабатьки» сучасних ігор. Так само як патент на використання ЕПТ (електронно-променевої трубки) в ігрових цілях (1947 рік), створення алгоритму шахової гри для комп'ютера (1948 рік), а потім написання першої подібної програми під назвою «TUROCHAMP» (комп'ютерів, здатних її запустити тоді ще не було) в 1950-1951 роках.

Спроби створити простенькі ігри на цифрових пристроях робилися ще за роки Другої Світової війни (а в 1947 вже була запрограмована перша електронна гра, монітором для якої служив екран військового радара) - це був симулятор ворожих ракет — проте вважається, що першою комп'ютерною грою стала "ОХО "(" Хрестики нулики "), в поодинці зроблена А.С. Дугласом в далекому 1952 році.

Рисунок 1.1 – Емуляція першої гри

У 1958 році вченим Вільямом Хігінботемом був створений перший симулятор тенісу під назвою «Tennis for Two». Грати в нього могли двоє людей: вони управляли рухливими платформами, якими відбивали м'ячик.

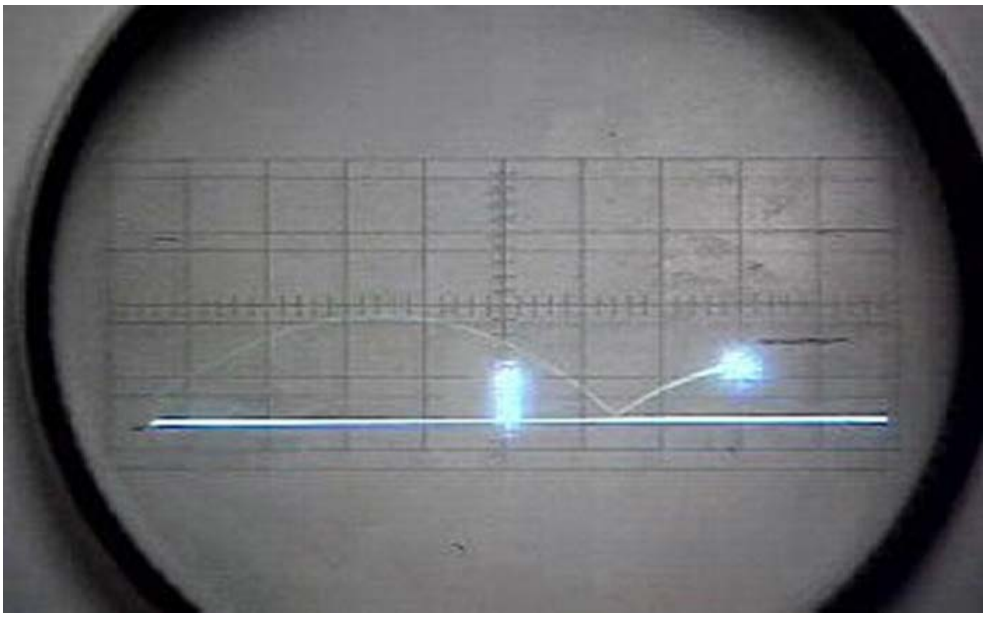

Рисунок 1.2 – Фрагмент гри Tennis for Two

Першими серед інших комп'ютерні ігри стали масово створювати оператори ЕОМ в різних напівзакритих наукових установах, що мали в розпорядженні ЕОМ. Наприклад, в 1962 році група студентів Массачусетського технічного інституту створила гру Spacewar! для нової ЕОМ DEC PDP-1. У Spacewar! гравцям надавалася можливість покерувати за допомогою контролерів космічними корабликами, що вистрілює один в одного ракетами. Складності грі додавало те, що кораблі кружляли навколо смертоносної чорної діри, розташованої в центрі ігрового поля. Spacewar! стала дуже популярна по всьому світу від того, що компанія DEC вбудувала її у свої комп'ютери як програму, за допомогою якої перевірялася їх працездатність.

Рисунок 1.3 – Перший прототип гри Spacewar

Вже в 1966 році Ральф Баєр створив першу "телевізійну" гру Chase, що стала першою грою, пограти в яку можна було, вивівши картинку на екран телевізора у себе вдома. Також в 1967 вийшло кілька інтерактивних ігор, для яких був необхідний променевий пістолет.

У 1969 році програміст Кен Томпсон створив гру для ОС MULTICS. Вона була названа Space Travel. В ST гравець міг подорожувати на космічному кораблі між планет невеликої сонячної системи та посадити корабель на одну з них, що і було метою гри.

Але після закриття проекту MULTICS, Кену довелося шукати новий комп'ютер для своєї гри, яким стала ЕОМ компанії General Electric. Але працювати за нею було дуже дорого, тому Томпсон портував свою гру на старенький PDP-7. Для портування гри на цей комп'ютер і була створена ОС UNIX. Крім того, 1970 рік став роком народження комп'ютерної миші (автора — Дуглас Енгельгардт). Втім, в іграх її стали використовувати набагато пізніше.

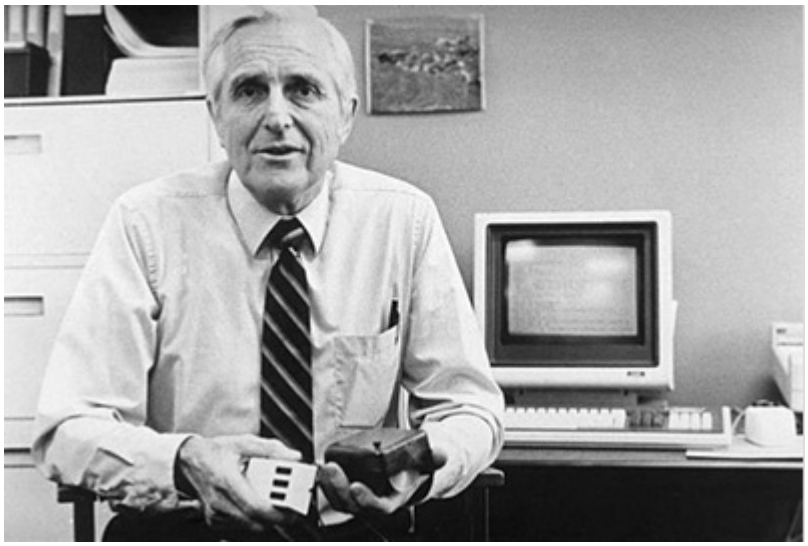

Рисунок 1.4 – Дуглас Енгельгардт зі своїм винаходом

У 1971 році гру Spacewar сильно спростили, прибравши з неї фізику та інерцію, що дало можливість зробити її набагато менш вибагливою до "заліза" і, помістивши її на компактні електронні плати, під'єднати їх до телевізора та отримати перший ігровий автомат. Було випущено півтори тисячі таких ігрових автоматів, що ознаменувало собою вихід комп'ютерних ігор в маси. Грати в такі автомати люди могли в громадських місцях — зрозуміло, за гроші.

У 1972 році Дебні і Бушнелл заснували ігрову компанію Atari, що створила незабаром стала популярну гру PONG, багато запозичили з Tennis for Two. Компанії вдалося продати понад 1900 автоматів з вшитою в них грою PONG. Було продано 19 000 ігрових автоматів. Так PONG стала найпершою за всю історію комп'ютерною грою, що багато разів окупилась.

Рисунок 1.5 – Ігровий автомат для гри в PONG

У тому ж році в Стенфордському університеті був проведений перший кіберспортивний турнір з SpaceWar. Цю дату можна вважати вихідною точкою зародження E-sports.

З 1971 по 1980 рік вийшли два шутера від першої особи на EOM - Maze War і SpaSim (два перших шутера, перший — в лабіринті від першої особи, другий — на тлі космосу), ще — перша текстова гра-квест Adventure у всесвіту D & D (Dungeons & Dragons), перша гра-квест Colossal Cave Adventure. Були створені перші прототипи мережевих ігор, з'явилися картинки з текстових символів (ASCII-art).

Друге покоління відеоігор (1976-1983)

У 1976 році були створені портативні носії інформації (ROM-катріджі), що дозволило не зашивати по одній грі в один комп'ютер, а записувати ігри на катріджі, вставляти їх в спеціальні слоти та грати на одній машині в різні комп'ютерні ігри. Такі комп'ютери стали називати консолями. Першою консоллю в результаті стала консоль VES 1976 року випуску від Fairchild, але вона не стала

особливо популярною, на відміну від VCS, Intellivision, ColecoVision. У 1977 році Стів Джобс з товаришами випускають комп'ютер для домашнього користування Apple II, який також став платформою для створення ігор на комп'ютери. [5]

Рисунок 1.6 – Комп’ютер Apple 2 1977 року

Ігрова консоль VCS (Video Computer Sistem (Atari 2600)) вийшла в 1977 році та стала найпопулярнішою з перших консолей. Вона володіла 8-ми бітовим процесором. Наступною консоллю була Intellivision 1980-го року, яка відрізнялася більш потужним, ніж у VCS, десяти-бітним процесором, і більш продуктивним відеочіпом. За 7 років (1977-1983) було продано понад сорок мільйонів екземплярів консолі Atari 2600.

Рисунок 1.7 – Ігрова консоль Atari 2600 1977 року

У 1979 році також вийшли в маси ігрові автомати Asteroids і Pac-Man. Після виходу Pac-Man ігрові автомати стали з'являтися в місцях скупчення людей (торгових центрах і т.д.).

Перша кишенькова приставка для ігор з'явилася в 1979 відомої американської компанії Milton Bradley, що спеціалізується на настільних іграх. Цей пристрій одержав назву Microvision. Він міг похвалитися монохромним квадратним рідкокристалічним екраном 16 на 16 пікселів. Під Mcrovision було створено дванадцять ігор (по два кілобайти кожна). Консоль ColecoVision з'явилася в 1982 році. Вона була ще потужніше, ніж її попередниці, але через монополію трьох основних виробників консолей ринок незабаром неабияк обвалився.

Третє покоління відеоігор (1982-1988)

У 1980 році за створення простих консолей з монохромним екраном взялася японська компанія Nintendo. Вона поставила перед собою мету — створити компактні пристрої для ігор в подорожах. В результаті з'явилася серія ігор Game and Watch. Саме їх прикладом надихалися радянські розробники при створенні комп'ютерних ігор фірми "Електроніка" (наприклад, "Ну, постривай!")

Рисунок 1.8 – Ігрова консоль від фірми "Електроніка"

У 1982 році на ринку з'явився Commondore 64 одночасно з англійським восьмирозрядним комп'ютером Sinclair Zx Spectrum. Останній став воістину легендарним — швидко поширившись по всьому світу, він продовжував мати попит ще більш як десять років.

Рисунок 1.9 – Sinclair Zx Spectrum 1982 року

Коли на ринку з'явилися IBM-комп'ютери, ігри знову стали ставати популярними. Нові технології, серед яких — нові звукові чіпи, 16-колірний (а згодом — і 256-кольорової VGA) стандарт дозволили розробникам ігор створювати більш складні й красиві ігри на персональні комп'ютери. [3]

У 1984 в Радянському Союзі з'явилася гра Tetris (Тетріс).

Однак і розробники ігрових консолей тим часом не сиділи склавши руки — в 1985 році побачила світ восьми-бітна ігрова консоль NES (Nintendo Entertainment System), що стала іконою галузі на десяток років. Всім вам відома гра Super Mario Brothers спочатку була випущена саме на консолі NES.

Четверте покоління відеоігор (1988-1994)

Цей період часу ознаменувався різким стрибком якості, кількості та здешевлення вартості як розробки комп'ютерних ігор, так і персональних комп'ютерів, причому в останніх почали з'являтися CD-ROM. В іграх почав з'являтися якісний звук, приємна оку графіка і спецефекти. У цей час утворилися такі гейм-дев компанії, як: Blizzard, Epic Games, id Software, Electronic Arts, 3D Realms і багато інших. Серед безлічі випущених в цей період часу ігор варто окремо виділити Mortal Combat, Street Fighter, Heroes of Might and Magic, Myst, Wolfenstein 3D, DUNE-2 (перша покрокова стратегія), Alone in the Dark (один з перших хорроров) DooM, King's Bounty (прабатько всіх сучасних RPG) і, звичайно, Civilization.

Рисунок 1.10 – Dune II 1992 року

Серед консолей цей період ознаменувався випуском в 1989 році 16-ти бітної Sega Mega Drive, на що Nintendo через два роки відповіла своїй SNES (те саме, що

і NES, але з префіксом "Супер"). Були й інші консолі, такі як Neo-Geo, PC Engine та інші, але вони не здобули такої популярності у публіки.

П'яте покоління відеоігор (1994-1999)

Поступово 2D гри йшли в минуле — їх місце потроху займали ігри з більш реалістичною 3D графікою. З'явилися 32-х бітні процесори. Знаковим став вихід в 1993 році Atari Jaguar, через рік після якого на ринок вийшли Sony PlayStation і Sega Saturn. Ігор стало ще більше — до того ж з випущеними попередньо іграми, багато з яких стали культовими, з'явилися гри Tekken, Resident Evil, Silent Hill, GTA, Need for Speed, Starcraft, Half- Life і т.д. На ринку консолей продовжували боротися троє виробників приставок - Sony, Nintendo і Sega. У ПК — відеоприскорювачів з'явилися стандарти DirectX і OpenGL - і, як наслідок цієї події, шутери від першої особи Unreal і Quake. [5]

Рисунок 1.11 – Quake 1996 року

Шосте покоління відеоігор (1999-2005)

У 1998 році вийшла потужна консоль Sega Dreamcast, яка через низку причин не змогла міцно закріпитися на ринку і незабаром впала з нього, потягнувши за собою і свого творця, Sega. Sony і Nintendo закріпилися на ринку за допомогою, відповідних: PlayStation 2 і Game Cube. Microsoft теж вирішила вставити свої п'ять копійок і випустила у 2001 році свою першу консоль Xbox.

Одночасно стали розвиватися гри на мобільних телефонах, в більшості своїй — на платформах Symbian, Java і Windows Mobile.

Рисунок 1.12 – TES 3 Morrowind 2002 року

Тим часом гри на PC розвивалися нітрохи не менш інтенсивно — з'явилося багато різних конфігурацій доступних звичайному споживачеві комп'ютерів, чому серед гравців стався розкол — ті, хто вважали за краще не витрачати гроші на потужні PC, грали в різноманітні квести, аркади, ребуси та головоломки, а ті, хто розщедрився на потужний PC - насолоджувалися дуже красивими та вимогливими до заліза іграми, такими як Quake 3, Half-Life 2, Morrowind, Age of Empires, Unreal Tournament 2004, GTA: San Andreas і багатьма іншими. Процесори стають все потужніше, на ринок вийшли ATI, AMD і NVidia, потіснивши Intel. У все більшої кількості людей почав з'являтися доступ до Інтернету, що посприяло розвитку низькобюджетних інді-студій по розробці ігор, які отримали можливість продавати свої ігри через Інтернет без витрат на покупку і перепродаж фізичних носіїв. В той самий час потужно розвинулося цифрове піратство, особливо в країнах СНД.

Рисунок 1.13 – Age of Empires 2 1999 року

Разом з цим починає набирати популярність «Dota» - гра, що представляє модифікацію однієї з карт «Warcraft III». Надалі ця карта оформилася в окрему гру, а у 2013 році вона отримала продовження і стала популярною кіберспортивною дисципліною під назвою «Dota 2».[4]

Рисунок 1.14 – Комп’ютерна гра Dota 2

Сьоме покоління відеоігор (2005-2015)

Ігри стають все більш вимогливим до заліза, з'являються нові консолі, багатоядерні процесори, Інтернет є практично у всього населення, продажі ігор стрімко переходять в цифровий формат, хоч і виходять об'ємні Blu-ray диски, які, проте, не змогли виграти боротьбу із дешевшими та більшими в розмірі флеш-картами. Під кінець в конструкцію багатьох нових ноутбуків навіть перестали додавати оптичний привід — заради економії місця, зменшення і полегшення корпусу.

Найбільші розробники стали випускати кросплатформені проекти відразу на різні ігрові платформи для кращої окупності, стало з'являтися все більше інді-студій, що черпають кошти на краудфандингу (збір пожертвувань у потенційних гравців на спеціально створених для цього майданчиках). Кількість нових ігор, що випускаються щороку, росте лавиноподібно. [5]

2012 рік. Була випущена «The Witcher 3: Wild Hunt» - найкраща РПГ за мотивами слов'янського фентезі.

Рисунок 1.15 – Комп’ютерна гра The Witcher 3: Wild Hunt

Восьме покоління відеоігор (2015- наш час)

Приблизно у 2014-2015 році на ринку віртуальних розваг з'явилась технологія VR. Virtual Reality почала пробиватися відразу по всіх фронтах — і на ринок портативних мобільних пристроїв, і на ринок стаціонарних ПК і ігрових консолей. Але у віртуальній реальності виникло безліч проблем, серед яких недостатня кількість ігрових проектів, що підтримують технологію VR, висока

вартість пристроїв для неї й сирі технології взаємодії з віртуальним світом (різні джойстики, сенсори та камери або забезпечували незручну або просто погану взаємодію з віртуальністю, або коштували дуже багато). Серед VR-шоломів можна виділити: Oculus Rift, HTC Vive, Sony PlayStation VR, Samsung Gear VR і картонний Google Cardboard (два останні — для смартфонів).

Через кілька років варто очікувати сплеску інтересу до цієї технології — коли більшість пересічних користувачів будуть мати потужні ПК і будуть готові купувати гри з підтримкою шоломів віртуальної реальності, провідні виробники ігор напевно відреагують на це випуском нових ігор AAA класу з підтримкою VR.

Відеоігри вже давно стали частиною сучасної культури: у багатьох країнах вони вже офіційно визнані мистецтвом. Кіберспорт набирає популярність і за прогнозами аналітиків аудиторія глядачів, а також фінансовий оборот цієї галузі вже до 2025 року перевищить стандартні види спорту. [5]

1.3 Жанри комп’ютерних ігор

Кількість комп'ютерних ігор просто величезна, і часто вони класифікуються за характеристиками або завданням. Тому категорії ігор, або жанри, можуть поділятися на піджанри, а одна гра цілком може належати до кількох жанрів.

Так ігрові рушії, як правило, специфікуються під жанр ігор. Рушій, призначений для гри в файтинг для двох осіб у боксерському рингу, буде дуже відрізнятися від багатокористувацької онлайн-ігри (MMOG) або двигуна шутерів від першої особи (FPS) або двигуна стратегії реального часу (RTS). Однак також існує велика кількість накладень — всі 3D-ігри, незалежно від жанру, вимагають певної форми введення керування грою з джойстика, клавіатури та / або миші, певної 3D рендеру, деякої форми дисплею ігрової інформації (HUD), що включає візуалізацію тексту в різних шрифтах, потужну аудіосистему і так далі.

Шутери від першої особи (FPS)

Жанр шутера від першої особи (FPS) вводиться такими іграми, як Quake, Unreal Tournament, Half-Life, Counter-Strike та Call of Duty. Ці ігри історично

включали порівняно повільний пішохідний перехід потенційно великого, але насамперед коридорного світу. Однак сучасні шутери від першої особи можуть проводитись у найрізноманітніших віртуальних середовищах, включаючи великі відкриті та конфініковані приміщення. Сучасна механіка руху FPS може включати в себе пішохідний рух, можливість перевезення гравця через наземний транспорт або повітряний транспорт, і навіть через наводний транспорт.

Шутери від першої особи, як правило, фокусуються на таких технологіях, як:

- ефективна візуалізація великих 3D-віртуальних світів;
- точних механік управління / прицілювання камери;
- анімації з високим рівнем деталізації віртуальної зброї та рук гравця;
- широкий асортимент потужного ручного озброєння;
- модель поведінки камери, що згладжує різкість зіткнення гравця з різноманітними об'єктами;
- великий набір анімації та пророблений інтелект для не ігрових персонажів (вороги та союзники гравця);
- можливість гри через інтернет Інтернеті (як правило, сервери підтримують до 64 гравців одночасно) та класичний для цього жанру режим гри «Deathmatch»

Платформери та інші ігри від третьої особи (3RD PERSON)

«Платформер» - це термін, що застосовується до персонажних екшн-ігор від третьої особи, де стрибки з платформи на платформу є основним механізмом ігрового процесу. Типові ігри з епохи 2D включають Space Panic, Donkey Kong, Pitfall! Та Super Mario Brothers. Епоха 3D включає такі платформи, як Super Mario 64, Crash Bandicoot, Rayman 2, The Hedgehog, серії Jak and Daxter, серії Ratchet & Clank та останнім часом Super Mario Galaxy.

Ігри на основі персонажів від третьої особи мають багато спільного з іграми від першої особи, як шутери, але значно більше уваги приділяється здібностям персонажів та руху персонажа в грі. Крім того, для персонажа потрібні

високоякісні анімації для всього тіла, на відміну від дещо менш вибагливих вимог до анімації "floating arm" у типовій грі FPS. Важливо відзначити, що майже всі шутери від першої особи мають онлайн-компонент для гри з іншими гравцями, тому аватар гравця з повним тілом повинен бути наданий на додаток до анімації рук від першої особи.

У платформери головний герой часто подібний до мультфільмів і не особливо реалістичний або не з високою роздільною здатністю. Однак шутери від третьої особи часто мають дуже реалістичної гуманоїдної моделі гравця. В обох випадках персонаж гравця, як правило, має дуже багатий набір дій та анімації. Деякі з технологій, спеціально орієнтовані на ігри в цьому жанрі, включають:

- платформи, що рухаються, драбини, мотузки, та інші цікаві режими руху;
- головоломки з різними об'єктами у світі;
- камера в режимі від третьої особи залишається зосередженою на персонажі гравця та обертанням якої, як правило, керує гравець за допомогою палички джойстика (на консолі) або миші (на ПК);
- складна система уникнення зіткнення камери з об'єктами у світі, для забезпечення того, щоб точка огляду ніколи не «пробивалась» через геометрію фону або динамічні об'єкти переднього плану.

Файтинги

Файтинги або ігри про боротьбу, як правило, розраховані для двох гравців, в яких персонажі гравців б'ють один одного на рингу. Жанр типізований такими іграми, як Soul Calibur та Tekken. Традиційно ігри в жанрі файтингу зосередили свою технологію на таких речах як:

- великого набору анімацій ударів для персонажів;
- точне виявлення ударів;
- система введення користувача, здатна виявляти складні комбінації кнопок і джойстиків;
- натовп або відносно статичний цікавий фон.

Оскільки тривимірний світ у цих іграх малий і камера постійно зосереджена на двох гравцях, історично в цих іграх мало або взагалі не потрібно світового підрозділу або відключення оклюзії. Так само не використовуються передових тривимірних моделей розповсюдження звуку.

Гонки

Жанр гонок охоплює всі ігри, основним завданням яких є керування автомобілем чи іншим транспортним засобом на якійсь трасі. У жанрі є багато підкатегорій. Готові ігри, орієнтовані на імітацію («симуляцію»), мають на меті забезпечити максимально реалістичний досвід водіння (наприклад, Gran Turismo). Аркадні гонки віддають перевагу забаві над реалістичністю (наприклад, San Francisco Rush, Cruisin 'USA, Hydro Thunder). Порівняно новий піджанр досліджує субкультуру вуличних перегонів із викрадених транспортних засобів (наприклад, Need for Speed, Juiced). Картингові гонки — це підкатегорія, в якій популярні персонажі з платформерних ігор або персонажі мультфільмів з телевізора перетворюються на роль водіїв картингових транспортних засобів (наприклад, Mario Kart, Jak X, Freaky Flyers).

Гоночна гра часто дуже лінійна, подібно до старих ігор FPS. Однак швидкість подорожі, як правило, набагато швидша, ніж у FPS. Тому більше уваги приділяється дуже довгим коридорам доріжок, або петель, які іноді мають різні альтернативні маршрути та таємні скорочення. Гоночні ігри зазвичай зосереджують усі свої графічні деталі на транспортних засобах, дорозі та найближчому оточенні. Проте картингові гонки також приділяють значну частину візуалізації на анімацію для персонажів, що керують транспортними засобами. Деякі технологічні властивості типової гоночної гри включають такі методи:

- Різні виду рендерінгу використовуються під час надання віддалених фонових елементів, наприклад, використання двовимірних обьєктів для дерев, пагорбів та гір.
- Дорога часто розбивається на відносно прості двовимірні регіони, які називаються "секторами". Ці структури даних використовуються для оптимізації

візуалізації та визначення видимості, для побудови маршуту для автомобілів, що не контролюються людьми, та для вирішення багатьох інших технічних проблем.

- Камера, як правило, стоїть позаду транспортного засобу з точки зору третьої особи, або іноді знаходиться всередині кабіні.

Стратегія в реальному часі (RTS)

Жанр сучасної стратегії в реальному часі (RTS) був визначений грою Dune II: Будівництво династії (1992). Інші ігри цього жанру включають Warcraft, Command & Conquer, Age of Empires та Starcraft. У цьому жанрі гравець стратегічно розгортає бойові підрозділи у своєму арсеналі через велике ігрове поле, намагаючись перемогти свого опонента. Ігровий світ зазвичай відображається під косим кутом огляду зверху вниз. Гравцю RTS гри, зазвичай заважають змінювати кут огляду, щоб бачити на великій відстані. Це обмеження дозволяє розробникам використовувати різні оптимізації в механізмі візуалізації гри RTS. Старіші ігри в жанрі використовували структуру світової побудови на основі сітки, а орфографічна проєкція була використана для того, щоб значно спростити рендерінг. Сучасні ігри RTS іноді використовують перспективу та справжній тривимірний світ, але вони все ще не можуть не використовувати систему компонування сітки, щоб забезпечити, коректне розміщення бойових одиниць та фонові елементи, такі як будівлі, щоб вони правильно відповідали одна одній. Деякі поширені практики в іграх RTS включають такі методи:

- Кожен об'єкт має невелику кількість полігонів, так що гра може підтримувати велику кількість їх на екрані одночасно.
- Рельєф місцевості, як правило, має різні рівні, на яких гравці мають більш або менші переваги перед опонентом.
- Гравець часто дозволяється будувати нові споруди на місцевості на додаток до розгортання своїх сил.
- Взаємодія користувачів, як правило, здійснюється за допомогою одного клацання та вибору одиниць на основній області, на додаток є меню або

панелі інструментів, що містять команди, обладнання, типи підрозділів, типи будівель тощо.

Ігри з масовим багатокористувацьким онлайном (MMOG)

Жанр масової багатокористувацької онлайн-ігри (MMOG) типізується такими іграми, як Neverwinter Nights, EverQuest, World of Warcraft та Galaxies Star Wars Galaxies. MMOG визначається як будь-яка гра, яка підтримує величезну кількість гравців водночас на сервері (від тисяч до сотень тисяч), як правило, всі вони грають в одному дуже великому віртуальному світі (тобто світі, внутрішній стан якого зберігається протягом дуже тривалих періодів часу, набагато вищий за сеанс ігрового процесу будь-якого гравця). В іншому випадку досвід гри MMOG часто схожий на досвід невеликих мало користувацьких ігор. До підкатегорій цього жанру належать рольові ігри MMO (MMORPG), стратегічні ігри MMO в режимі реального часу (MMORTS) та шутери від першої особи MMO (MMOFPS).

В основі всіх MMOG - дуже потужна батарея серверів. Ці сервери підтримують стан ігрового світу, керують користувачами, які входять і виходять із гри, надають чати або послуги з передачею голосу через IP (VoIP) тощо. Майже всі MMOG вимагають, щоб користувачі платили якусь регулярну абонентську плата, щоб грати, і вони можуть робити мікро-транзакції всередині ігрового світу або поза грою. Отже, найважливіша роль центрального сервера полягає в обробці рахунків та мікро-транзакцій, які служать основним джерелом доходу розробника ігор.

Графічна здатність у MMOG майже завжди нижча, ніж ігор аналогів інших жанрів, внаслідок величезних розмірів ігрового світу та надзвичайно великої кількості користувачів онлайн, підтримуваних подібними іграми.

Інші жанри

- Звичайно, існує безліч інших жанрів ігор. Деякі приклади:
- спорт, з піджанрами для кожного основного виду спорту (футбол, бейсбол, футбол, гольф тощо);
- Ігри Бога, як Populus і Black & White;

- екологічні / соціальні імітаційні ігри, як SimCity або The Sims;
- ігри-головоломки на зразок тетрісу;
- перетворення неелектронних ігор в електронний варіант, як-от шахи, ігри в карти, тощо;
- веб-ігри, такі, що створені на веб-сайті Electronic Arts 'Pogo;
- і список продовжується.

Кожен ігровий жанр має свої особливі технологічні вимоги. Це пояснює, чому ігрові рушії традиційно відрізняються від жанру до жанру. Однак також існує велика кількість технологічних перекриттів між жанрами, особливо в контексті єдиної апаратної платформи. Саме тому під кожний жанр ігор, потужні компанії створюють власні закриті ігрові рушії, для своїх ААА ігор та проектів, оскільки неможливо використати один і той же рушій, для різних жанрів ігор, при цьому не знизивши показники оптимізації або графіки.

1.4 Висновки до розділу 1:

У даному розділі було розкрито такі аспекти іігрової індустрії:

- Визначення комп'ютерної гри;
- Історія розвитку комп'ютерних ігор;
- Характеристики компьютерных ігор ;
- Перелік основних жанрів комп’терних ігор;
- Вимоги до створення ігор певних жанрів;

2 ТЕХНІЧНІ АСПЕКТИ РЕАЛІЗАЦІЇ ІГРОВИХ РУШІЇВ

2.1 Визначення ігрового рушія

Ігровий рушій - базове програмне забезпечення комп'ютерної гри. Поділ гри і ігрового рушія часто розмите, і не завжди студії проводять чітку межу між ними. Але в загальному випадку термін «ігровий рушій» застосовується для того програмного забезпечення, яке придатне для повторного використання і розширення, і тим самим може бути розглянуто як підставу для розробки безлічі різних ігор без істотних змін.[7]

Термін «Ігровий рушій» виник у середині 1990-х у відношенні до шутерів від першої особи (FPS), таких як Doom від компанії id Soft. Doom був розроблений з досить чітко визначеним поділом між основними компонентами програмного забезпечення (наприклад, тривимірна система візуалізації графіки, система виявлення зіткнень або аудіосистема) та об’єктами, ігровими світами та правилами гри які включали в себе ігровий досвід гравця. Значення цього розмежування стало очевидним, коли розробники почали ліцензувати ігри та перевлаштовувати їх у нові продукти, створюючи нові ігрові об’єкти, карту світу, зброю, персонажів, транспортні засоби та правила гри, лише зі змінами програмного забезпечення. Це ознаменувало народження спільноти розробників модифікацій — групи окремих геймерів та невеликих незалежних студій, які будували нові ігри, модифікуючи існуючі ігри, використовуючи безкоштовні набори інструментів, надані оригінальними розробниками.

Наприкінці 90-х років деякі ігри, такі як Quake III Arena та Unreal, були розроблені з урахуванням повторного використання та «відхилення». Рушії були налаштовані за допомогою мов скриптів, таких як Id Quake C, і ліцензування двигунів стало додатковим потоком доходів для розробників, які їх створили. Сьогодні розробники ігор можуть ліцензувати ігровий движок та повторно використовувати значні частини своїх основних компонентів програмного забезпечення для створення ігор. Хоча ця практика все ще передбачає значні

інвестиції в інженерну розробку програмного забезпечення, вона може бути набагато економічною, ніж розробка всіх основних компонентів двигуна.

Межа між грою та її двигуном часто розмита. Деякі двигуни досить чітко розрізняють цю межу, в той час як інші майже не мають змоги розділити ці два поняття. В одній грі код візуалізації може спеціально «знати», як намалювати один певний об’єкт. В іншій грі механізм візуалізації може надавати матеріали загального призначення та засоби для затінення, що дозволить створити безліч різноманітних об’єктів. Жодна студія не робить абсолютно чіткого розмежування між грою та двигуном, що зрозуміло, враховуючи, що визначення цих двох компонентів часто змінюються в міру того, як визначений дизайн гри. Можливо, архітектура, керування даними — це те, що відрізняє ігровий движок від програмного забезпечення, яке є грою, але не двигуном. Коли гра містить жорстко закодовану логіку, чи правила гри або використовує спеціальний код для візуалізації конкретних типів ігрових об’єктів, використовувати це програмне забезпечення для створення іншої гри стає важко або неможливо.

Тож термін «ігровий рушій» можна визначити, як програмного забезпечення, що розширюється і може використовуватися як основа для багатьох різних ігор без великих змін.

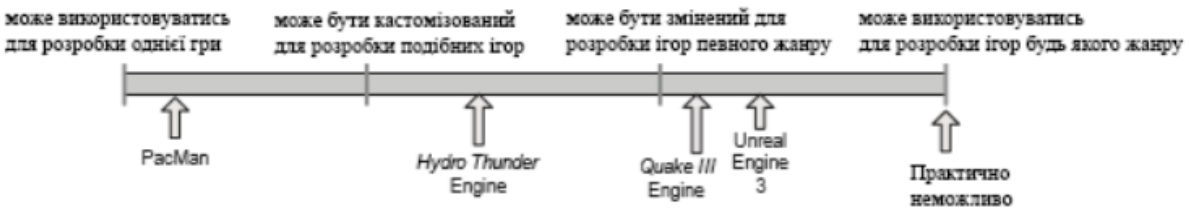

Рисунок 2.1 – Гнучкість ігрових рушіїв

На рис 2.1 зображено еволюція гнучкості ігрових рушіїв, хоча і на сьогоднішній день існують готові рішення для гнучких ігрових рушіїв, однак все ж поки що неможливо використати один і той же рушій для всіх жанрів ігор.[6]

2.2 Архітектура ігрового рушія

Ігровий рушій зазвичай складається з набору інструментів і компонентів для виконання. На рис 2.2 показані всі основні компоненти, які складають типовий рушій 3D-ігор Як і всі програми, ігрові рушії побудовані в шари. Зазвичай верхні шари залежать від нижніх шарів, але не навпаки.

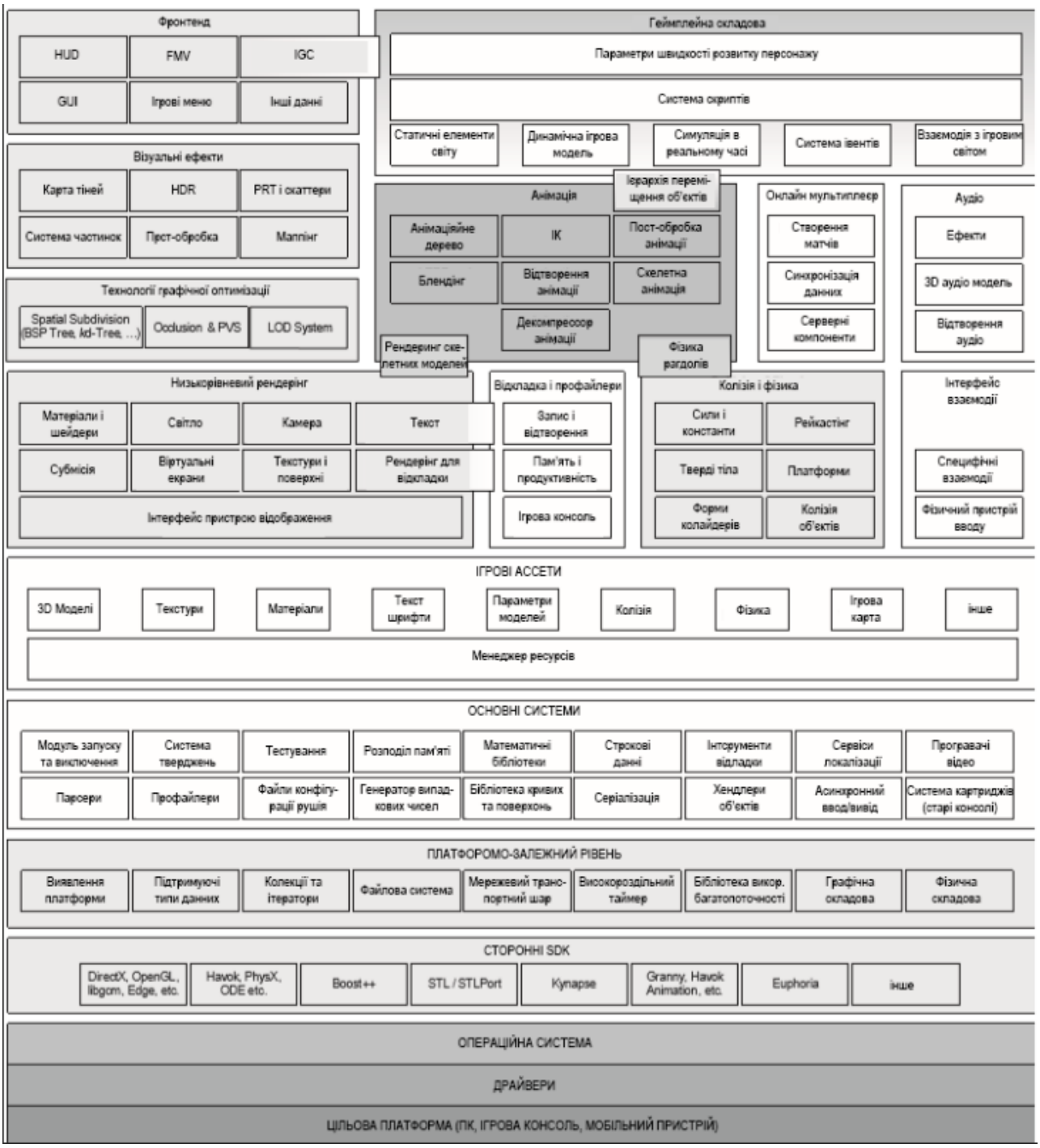

Рисунок 2.2 – Основні елементи ігрових рушіїв

Короткий огляд компонентів, показаних на рис 2.2:

Цільовий апаратний рівень, являє собою комп'ютерну систему або консоль, на якій буде працювати гра. Типові платформи включають ПК на базі Microsoft Windows і Linux, Apple iPhone і Macintosh, Microsoft Xbox і Xbox 360, Sony PlayStation, PlayStation 2, PlayStation Portable (PSP) і PlayStation 3, а також DS, GameCube і Wii Nintendo.

Драйвери пристроїв — це компоненти низького рівня програмного забезпечення, що надаються операційною системою або постачальником обладнання. Драйвери керують апаратними ресурсами та захищають операційну систему та верхні шари двигуна від неправильного використання системних ресурсів з безліччю наявних варіантів апаратних пристроїв.

Операційна система (ОС) – вона керує виконанням декількох програм на одному комп’ютері, однією з яких є гра. Операційні системи, такі як Microsoft Windows надає певну кількість апаратних ресурсів кожній програмі, яка запущена в системі, дана функція називається багатозадачністю. Це означає, що комп'ютерна гра ніколи не може вважати, що вона повністю контролює обладнання - вона повинна «ладити» з іншими програмами в системі.

На консолі операційна система часто є лише тонким шаром бібліотеки, який збирається безпосередньо у вашій грі. На консолі гра зазвичай «володіє» всією машиною. Операційна система на цих консолях може зупинити виконання вашої гри або зайняти певні системні ресурси для того, щоб виводити повідомлення в Інтернеті або дозволити гравцеві зупиняти гру та вивести панель Xross Media PS PS3 або панель приладів Xbox 360.

Сторонні SDK. Більшість ігрових двигунів використовують ряд сторонніх наборів для розробки виробів (SDK) та середнього програмного забезпечення. Функціональний або класифікований інтерфейс, що надається SDK, часто називається інтерфейсом програмування для додатків (API).

Структури даних та алгоритми.

Як і будь-яка система програмного забезпечення, ігри в значній мірі залежать від структур даних та алгоритмів збору даних. Ось кілька прикладів сторонніх бібліотек, які надають такі послуги. STL.

Стандартна бібліотека шаблонів C ++ забезпечує безліч кодів і алгоритмів для управління структурами даних, рядками та потоком вводу-виводу.

STLport. Це портативна, оптимізована реалізація STL.

Boost - це потужна структура даних та бібліотека алгоритмів, розроблена в стилі STL.

Loki - це потужна загальна бібліотека шаблонів програмування.

Розробники ігор поділяються на питанні, чи використовувати бібліотеки шаблонів типу STL у своїх ігрових двигунах. Деякі вважають, що паттерни розподілу пам'яті STL, які не сприяють високоефективному програмуванню і, як правило, призводять до фрагментації пам'яті, роблять STL непридатним у грі. Інші вважають, що потужність та зручність STL переважують її проблеми, і що більшість проблем насправді можна вирішити в будь-якому разі. STL добре підходить для використання на ПК, оскільки його розвинена система віртуальної пам’яті робить необхідність ретельного розподілу пам’яті трохи менш важливою. На консолі, з обмеженими можливостями віртуальної пам’яті або відсутністю надмірних кеш-помилок, ймовірно, краще писати власні структури даних, які мають передбачувані або обмежені паттерни розподілу пам’яті.

Графіка

Більшість двигунів рендерінгу ігор побудовані поверх бібліотеки апаратних інтерфейсів, таких як:

Glide - це 3D-графічний SDK для старих відеокарт Voodoo. Цей SDK був популярний до епохи апаратного перетворення та освітлення (апаратне T&L), яке починалося з DirectX 8.

OpenGL - широко використовуваний портативний 3D-графічний SDK.

DirectX - це 3D SDK для графічної графіки Microsoft і основний конкурент OpenGL.

libgcm - це низькорівневий прямий інтерфейс до графічного обладнання PlayStation 3, яке було надано компанією Sony як більш ефективна альтернатива OpenGL.

Edge - це потужний і високоефективний механізм візуалізації та анімації, який виробляли Naughty Dog та Sony для Playstation 3 та використовувались у ряді перших та сторонніх ігрових студій.

Колізія та фізика

Виявлення зіткнень та жорстка динаміка тіла (відома просто як "фізика" у спільноті розвитку ігор) забезпечуються наступними відомими SDK.

Havok - популярний фізичний двигун фізики та зіткнення.

PhysX - ще один популярний двигун фізики та зіткнення промислового виробництва, доступний для безкоштовного завантаження з NVIDIA.

Open Dynamics Engine (ODE) - добре відомий пакет з фізики та колізії з відкритим кодом.

Анімація персонажів

Існує ряд комерційних пакетів анімації, включаючи, але, безумовно, не обмежуючись цим.

опулярний інструментарій Granny інструментів Rad Game Tools включає надійні експортери 3D-моделей та анімації для всіх основних пакетів 3D-моделювання та анімації, як Maya, 3D Studio MAX тощо.

Анімація Havok. Межа між фізикою та анімацією стає все більш розмитою, оскільки персонажі стають все більш реалістичними. Компанія, яка робить популярний SDK Havok з фізики, вирішила створити безкоштовну SDK-анімацію, що дозволяє усунути розрив фізичної анімації набагато простіше.

Бібліотека Edge, створена для PS3 командою ICE в Naughty Dog, групі Інструменти та Технології Sony Computer Entertainment America, та Sony Advanced Group Group в Європі, включає потужний та ефективний механізм анімації та ефективний двигун обробки геометрії для візуалізації.

Штучний інтелект

Kynapse. Донедавна для кожної гри штучний інтелект (AI) оброблявся у власному порядку. Однак компанія під назвою Kynogon виробила програмне забезпечення SDK під назвою Kynapse. Цей SDK забезпечує власності, такі як визначення шляху, статичне та динамічне уникання об'єктів, ідентифікація вразливих місць у просторі (наприклад, відкрите вікно, з якого може потрапити засідка), і досить хороший інтерфейс між AI та анімацією.

Біомеханічні моделі персонажів

Endorphin and Euphoria. Це пакети анімації, які виробляють рух символів, використовуючи вдосконалені біомеханічні моделі реалістичного руху людини.

Платформо-залежний рівень

Більшість ігрових рушіїв повинні працювати на більш ніж одній апаратній платформі. Наприклад, такі компанії, як Electronic Arts і Activision / Blizzard, завжди націлюють свої ігри на найрізноманітніші платформи, оскільки це виставляє їх ігри на найбільший можливий ринок. Як правило, єдині ігрові студії, які не націлені на щонайменше дві різні платформи на гру, - це найперші студії, наприклад студії Naughty Dog та Insomniac від Sony. Тому більшість ігрових двигунів архітектурно розташовані із рівнем незалежності платформи,. Цей шар розташований поверх апаратного забезпечення, драйверів, операційної системи та інших сторонніх програмних засобів та захищає решту двигуна від більшості знань базової платформи. Обгортаючи або замінюючи найбільш часто використовувані стандартні функції бібліотеки C, виклики операційної системи та інші основні інтерфейси програмування прикладних програм (API), рівень незалежності платформи забезпечує послідовну поведінку на всіх апаратних платформах. Це необхідно, тому що існує велика кількість варіацій на різних платформах, навіть серед «стандартизованих» бібліотек, як стандартна бібліотека C.

Основні системи

Кожен ігровий двигун, складний програмний додаток C ++, вимагає використання корисних програм для програмного забезпечення. Ось кілька прикладів засобів, які зазвичай забезпечує основний шар.

Твердження - це рядки коду перевірки помилок, які вставляються, щоб зафіксувати логічні помилки та порушення початкових припущень програміста. Затвердження тверджень, як правило, позбавляються від остаточної складової гри.

Управління пам'яттю. Практично кожен ігровий механізм реалізує власну систему розподілу пам’яті, щоб забезпечити високошвидкісні розподіли та переміщення, а також обмеження негативних ефектів фрагментації пам’яті.

Математична бібліотека. Ігри за своєю природою дуже математичні. Отже, кожен ігровий двигун має принаймні одну математичну бібліотеку. Ці бібліотеки надають засоби для векторної та матричної математики, обертання кватерніона, тригонометрії, геометричних операцій з лініями, променями, сферами, тощо, маніпуляції з сплайном, чисельна інтеграція, розв’язування систем рівнянь та будь-яких інших засобів, які вимагають ігрові програмісти.

Спеціальні структури даних та алгоритми. Якщо розробники двигуна не вирішили повністю покластися на сторонній пакет, такий як STL, набір інструментів для управління основними структурами даних (пов'язані списки, динамічні масиви, бінарні дерева, хеш-карти тощо) та алгоритми (пошук, сортування, тощо), як правило, потрібно. Вони часто кодуються вручну для мінімізації або усунення динамічного розподілу пам'яті та для забезпечення оптимальної продуктивності виконання на цільовій платформі.

Менеджер ресурсів

Присутні в кожному ігровому рушії в певній формі, менеджер ресурсів забезпечує уніфікований інтерфейс (або набір інтерфейсів) для доступу до будь-яких і всіх видів ігрових активів та інших вхідних даних двигуна. Деякі двигуни роблять це централізовано та послідовно (наприклад, пакети Unreal, клас OGRE 3D ResourceManager). Інші двигуни застосовують спеціальний підхід, часто

залишаючи його програвачу ігор для прямого доступу до диску або в стислих архівах, таких як PAK-файли Quake.

Рушій рендерингу

Рушій візуалізації є одним з найбільших і найскладніших компонентів будь-якого ігрового двигуна. Візуалізатори можуть бути побудовані різними способами. Немає прийнятого способу зробити це, хоча, як ми побачимо, більшість сучасних рендеринг поділяють деякі фундаментальні філософії дизайну, що значною мірою визначається дизайном обладнання для графічної 3D-графіки, від якого вони залежать. Один поширений і ефективний підхід до проектування двигуна - використовувати шарувату архітектуру, як описано нижче.

Низькорівневий рендерінг

Низькорівневий візуалізатор, охоплює всі основні засоби візуалізації двигуна. На цьому рівні дизайн орієнтований на надання колекції геометричних примітивів якомога швидше і частіше, без особливого врахування, які частини сцени можуть бути видні. Цей компонент розбивається на різні підкомпоненти, про які йде мова нижче.

Інтерфейс графічного пристрою

Для графічних SDK, таких як DirectX і OpenGL, потрібна велика кількість коду, щоб перерахувати наявні графічні пристрої, ініціалізувати їх, налаштувати візуалізацію поверхонь (зворотний буфер, буфер трафарету тощо), і так далі. Це зазвичай обробляється компонентом, який я буду називати інтерфейсом графічного пристрою

. Для двигуна в іграх на ПК також потрібен код, щоб інтегрувати рендера в цикл повідомлень Windows. Зазвичай ви пишете "message pump", який обслуговує повідомлення Windows, коли вони очікують на розгляд, і в іншому випадку запускає цикл візуалізації знову і знову так швидко, як це можливо. Це пов'язує цикл опитування клавіатури гри з циклом оновлення екрана рендерінга. Це з'єднання небажане, але за допомогою певного ефекту можна мінімізувати залежності.

Інші компоненти рендерера

Інші компоненти рендерера низького рівня співпрацюють, щоб збирати подання геометричних примітивів (іноді їх називають пакетами візуалізації), такими як сітки, списки рядків, списки точок, частинки, виправлення місцевості, текстові рядки та все інше ви хочете намалювати та вивести їх якомога швидше. Низькорівневий візуалізатор зазвичай забезпечує абстракцію вікна перегляду з пов'язаною з ним матрицею-світовим матрицею та параметрами 3D-проекції, такими як поле зору та розташування найближчої та далекої площини кліпу. Низькорівневий рендер також керує станом графічного обладнання та шейдерів гри за допомогою своєї матеріальної системи та динамічної системи освітлення. Кожен представлений примітив асоціюється з матеріалом і впливає на n динамічних вогнів. Матеріал описує текстуруи, яку використовує примітив, те, які розстановки стану пристрою повинні бути чинними, і які вершинні та піксельні шейдери використовувати для надання примітиву. Фари визначають, як динамічні розрахунки освітлення застосовуватимуться до примітиву.

Графік сцени / Оптимізація відсікання

Низькорівневий рендерір малює всю представлену йому геометрію, не зважаючи на те, чи така геометрія видна чи ні. Компонент вищого рівня, як правило, необхідний для обмеження кількості примітивів, поданих на візуалізацію, виходячи з певної форми визначення видимості.. Для дуже маленьких ігрових світів - це просто відсікання (тобто видалення предметів, які камера не може "бачити"). Для великих ігрових світів може бути використана більш продвинута структура просторових підрозділів для підвищення ефективності візуалізації, дозволяючи дуже швидко визначати потенційно видимий набір (PVS) об'єктів. Просторові підрозділи можуть приймати різні форми, включаючи дерево розділення бінарного простору (BSP), квадрати, октагони , kd-дерево або ієрархію сфери. Просторовий підрозділ іноді називають графіком сцени, хоча технічно останній є особливим видом структури даних і не включає першого. Портали або методи відключення оклюзії можуть також застосовуватися в цьому шарі двигуна візуалізації. В ідеалі рендер низького рівня

повинен бути повністю агностичним щодо типу просторового підрозділу або графіка сцени, що використовується. Це дозволяє різним ігровим командам повторно використовувати примітивний код подання, але створити систему визначення PVS, яка є спеціальною для потреб гри кожної команди. Дизайн двигуна візуалізації з відкритим кодом OGRE 3D є чудовим прикладом цього принципу в дії. OGRE забезпечує архітектуру графічної сцени підключення та відтворення. Розробники ігор можуть або вибрати з декількох попередньо вдосконалених графічних конструкцій сцен, або вони можуть забезпечити реалізацію користувальницької сцени.

Візуальні ефекти

Сучасні ігрові двигуни підтримують широкий спектр візуальних ефектів,

- включаючи системи частинок (для диму, води, бризок води тощо);
- системи наклейки (для кульових отворів, відбитків ніг тощо);
- світлове відображення та картографування навколишнього середовища;
- динамічні тіні;
- Повноекранна публікація, що застосовується після того, як 3D-сцена була надана на другий буфер екрану.
- Деякі приклади повноекранних ефектів включають
- освітлення високого динамічного діапазону (HDR);
- повноекранне згладжування (FSAA);
- корекція кольору та ефекти зміщення кольорів, включаючи шунтування відбілювання, ефект насичення та знесилення кольорів тощо.

Ігровий двигун має системний компонент, який керує спеціалізованими потребами візуалізації частинок, наліпок та ін. візуальних ефектів. Системи частинок і відстійників зазвичай є різними компонентами двигуна візуалізації і виконують роль вхідних даних для низькорівневої візуалізації. З іншого боку, світлове відображення, картографування навколишнього середовища та тіні, як правило, обробляються всередині в межах власного механізму візуалізації.

Повноекранні ефекти або реалізуються як невід'ємна частина рендерінга, або як окремий компонент, який працює на вихідних буферах рендерінга.

Фронтенд

Більшість ігор використовують 2D-графіку, накладену на 3D-сцену для різних цілей. Сюди входить головний екран гри (HUD); меню в грі, консоль або інші інструменти розробки, які можуть поставлятися з кінцевим продуктом або не можуть бути поставлені; можливо, в грі графічний інтерфейс користувача (GUI), що дозволяє гравцеві маніпулювати інвентарем свого персонажа, конфігурувати підрозділи для ведення бою або виконувати інші складні ігрові завдання. Така двовимірна графіка, як правило, реалізується шляхом малювання текстурованих квадратиків (пар трикутників) з ортографічною проекцією. Або ж вони можуть бути виведені в повному 3D-форматі з накладеними на квадратики, щоб вони завжди стикалися з камерою. У цей шар включена система повнометражного відео (FMV). Ця система відповідає за відтворення повноекранних фільмів, які були записані попередньо.

Пов'язана з цим система - це система ігрових кінематографій (IGC). Цей компонент, як правило, дозволяє кінематографічні послідовності хореографувати у самій грі, у повному обсязі 3D. Наприклад, коли гравець гуляє містом, розмова між двома ключовими персонажами може бути реалізована як ігровий кінематограф. IGC можуть включати або не включати символів гравця. Вони можуть бути зроблені як навмисне відсічення, під час якого гравець не має контролю, або вони можуть бути тонко інтегровані в гру, без того, щоб гравець навіть не розумів, що відбувається IGC ролик.

Зіткнення та фізика

Виявлення зіткнення важливо для кожної гри. Без нього об’єкти би проникали один скрізь одного, і неможливо було б взаємодіяти з віртуальним світом будь-яким розумним чином. Деякі ігри також включають реалістичне або напівреалістичне моделювання динаміки. Вона називається «фізичною системою» в ігровій індустрії, хоча термін «жорстка динаміка тіла» дійсно доречніший, тому що зазвичай стосується лише рух (кінематика) жорстких тіл та сили і крутні

моменти (динаміка), які викликають це рух відбуватися. Зіткнення і фізика зазвичай досить щільно поєднані. Це тому, що коли виявляються зіткнення, вони майже завжди вирішуються як частина інтеграції фізики та логіки задоволення обмежень. В даний час дуже мало ігрових компаній пише власну двигун зіткнення та фізики. Натомість, SDK третьої сторони, як правило, інтегрується в двигун.

Havok - це золотий стандарт у сьогоднішній галузі. Він багатий на особливості та чудово працює на всіх платформах.

PhysX від NVIDIA - це ще один чудовий двигун зіткнення та динаміки. Він був інтегрований в Unreal Engine 3, а також доступний безкоштовно як окремий продукт для розробки ігор на ПК. Спочатку PhysX був розроблений як інтерфейс до нового мікросхеми прискорювача фізики Ageia. Тепер SDK належить і розповсюджується NVIDIA, і компанія адаптує PhysX для роботи на своїх останніх графічних процесорах.

Також доступна фізика з відкритим кодом та двигуни зіткнення. Мабуть, найвідомішим з них є движок Open Dynamics (ODE).

Анімація

Будь-яка гра, яка має органічних або напіворганічних персонажів (людей, тварин, мультиплікаційних персонажів або навіть роботів), потребує анімаційної системи. Існує кілька основних типів анімації, які використовуються в іграх:

спрайт / текстура анімації,

жорстка анімація ієрархії тіла,

скелетна анімація,

вершина анімація та

морфійські цілі.

Анімація скелету дозволяє аніматору створювати детальну сітку 3D-символів, використовуючи порівняно просту систему кісток. Коли кістки рухаються, вершини 3D-сітки рухаються разом з ними. Хоча морфологічні цілі та вершинна анімація використовуються в деяких двигунах, скелетна анімація є найбільш поширеним методом анімації в іграх сьогодні.

Відображення скелетної сітки - це компонент, який усуває розрив між візуалізатором та анімаційною системою. Тут відбувається тісна співпраця, але інтерфейс дуже чітко визначений. Анімаційна система створює позу для кожної кістки в скелеті, а потім ці пози передаються в механізм візуалізації як палітра матриць. Візуалізатор перетворює кожну вершину за допомогою матриці або матриць на палітрі, щоб генерувати остаточне змішане положення вершини. Цей процес відомий як скін. Існує також тісний зв’язок між системами анімації та фізики, коли використовуються ганчіркові ляльки. Ганчіркова лялька - це м'який (часто мертвий) анімований персонаж, рух тіла якого моделюється фізичною системою. Фізична система визначає положення та орієнтації різних частин тіла, трактуючи їх як обмежену систему жорстких тіл. Анімаційна система обчислює палітру матриць, необхідних механізму візуалізації для того, щоб намалювати символ на екрані. [6]

2.3 Готові рішення ігрових рушіїв

1. Unity 5 - крос-платформний ігровий рушій для розробки двомірних і тривимірних додатків та ігор під різні платформи. У Unity3d є дві версії: безкоштовна і платна. Відрізняються вони рядом можливостей, які можуть знадобитися при розробці гри. По-перше, безкоштовна версія Unity3d підтримує тільки Android, Web Player, PC-платформи. Повна версія дозволяє розробникам викладати своє творіння під всі найвідоміші платформи, такі як: PC, Linux, Mac, Windows Store, IOS, Android, Windows Phone 10 Store, Blackberry 10, Wii U, PS3, Xbox 360, PS4, Xbox One. Є можливість робити софт для VR (Virtual Reality), тобто під окуляри і шолом віртуальної реальності: Hololens, Oculus Rift, StarVR та інші, а також писати програми для Kinect 2.0, LeapMotion.

Можливості Unity 5

Unity3d має дуже простий Drag and Drop інтерфейс, який людина освоює за місяць. Весь рушій тільки на англійській мові. Русифікації Unity 5 немає. Unity розбитий на кілька вікон: Hierarchy, де знаходяться назви всіх об'єктів на сцені,

які можна групувати і легко переходити по ним, Scene, де можна розглянути певну сцену під потрібним вам ракурсом, Inspector, який допоможе з налаштуванням виділеного об'єкта, Project, де видно всі матеріали проекту, Toolbar (або меню з інструментами).

Unity 5 підтримує дві мови: C # (найбільш використовуваний) і Javascipt. Розробнику необхідно знати одну з мов досконало, а іншу на середньому рівні, так як деякі моменти Unity 5 робить тільки на одному з двох мов, або це робиться набагато важче, ніж на іншій мові програмування. Передостання версія Unity3d, а саме Unity 4, підтримувала мову програмування Boo (діалект Phyton), але його прибрали з 5-ої версії, так як їм практично ніхто не користувався, та й документації, на офіційному сайті Unity особливо не було. Розрахунки фізики в Unity 5 виконує NVIDIA PhysX. Зовсім недавно NVIDIA представила одну цікаву річ - NVIDIA Flex, яку, можливо, в майбутньому вмонтують в ігрові рушії.

Об'єкти в Unity3d можуть бути порожніми, (щоб об'єднати кілька об'єктів в одну групу, тобто зробити їх дочірніми GameObject), містити компоненти, з якими взаємодіють скрипти, можуть бути названі одним і тим же ім'ям, можуть бути присвоєні теги, які служать для того, щоб скрипт знайшов потрібний нам об'єкт. До об'єктів в Unity3d можна привласнити колайдери: Box Collider - куб, в який потрапляє модель об'єкта, Sphere Collider - сфера, Character Collider - коллайдер, який був спеціально введений в Unity 5 для використання під персонажів, Mesh Collider - коллайдер, створений за Мішу, тобто повторює геометрію об'єкта, Wheel Collider - коллайдер для коліс, Terrain Collider - коллайдер для Terrain - майданчики, яку використовують для відображення землі.

Анімувати моделі в Unity3d можна декількома способами: створення анімацій в спеціальних програмах, наприклад: 3Ds Max, Blender і інші, а можна і в самому Unity3d, так як редактор Unity має компонент для їх створення. Матеріали в Unity 5 відіграють важливу роль. Імпортовані текстури в Unity3d прикріпити до об'єкту не можна, необхідно створити матеріал, який можна привласнювати ігровому об'єкту. До призначеним матеріалу шейдерам будуть присвоєні текстури. Шейдери можна редагувати прямо в Unity3d. Unity 5 дозволяє

генерувати нормал-мапи (normal-map), лайт-мапи (light-map), різні альфа-канали та mip-рівні.

У повній версії Unity 5 можливо повне настроювання шейдеров, а в безкоштовній - немає.

Особливості Unity 5

Unity 5 має дві дуже важливі особливості: Occlusion Culling і Level Of Detail. Обидві речі дозволяють сильно знизити навантаження на центральний процесор, завдяки грамотній деталізації. Наприклад, в іграх жанру 2D і 3D Runner при подоланні певної дистанції все, що було позаду вас, видаляється, а то, що попереду вас, генерується. Таким чином, при тривалій грі ваше Пристрій не захаращує непотрібна інформація. Occlusion Culling не візуалізується геометрію і колайдери об'єктів, що знаходяться не в полі зорі камери, а Level Of Detail замінює деталізовані об'єкти, що знаходяться далеко від гравця, на менш деталізовані, причому розробник сам налаштовує цю систему. Тобто скромний проект може дозволити виставити величезні значення в Level of Detail, коли AAA-проекти виставляють його на мінімум.

Мінуси і плюси Unity 5

Unity 5 володіє величезною кількістю переваг перед іншими ігровими движками. Ком'юніті Unity 5 на сьогоднішній момент є найбільшим в світі. На офіційному сайті Unity є спеціальний розділ, в якому можна знайти статистику по ігровим движкам. За цими даними Unity 5 використовують понад 50% розробників відеоігор. 20% належать Unreal Engine, а решта ігрові движки - 30%. Для розробки 2D або 3D інді-ігор Unity 5 підходить за всіма параметрами. У Unity 5 дуже просто запікати проекти (білд). Причому можна створити один проект під безліч платформ, що дуже сильно полегшує процес для девелоперів. Всі скрипти, які використовуються в Unity 4, можна буде автоматично виправити в Unity 5.

Розробка AAA-проектів в Unity - найскладніший процес. По-перше, будь-який скрипт в Unity відразу тягне за собою купу помилок, яку в майбутньому необхідно виправити, або переписати скрипт заново. По-друге, все ще має погану оптимізацією. Весь контент, який стоїть у вас у вікні Project, але не варто у вас на

сцені, буде запечений, а значить, що гра буде важити в рази більше, ніж передбачалося. А найголовніше, що в інтернеті були питання про те, що проекти, в які не підключені до заводських значень, при білді все-одно запікати. Unity в найближчому оновленні пофиксил цей момент. У движку є ряд проблем зі скроллингом. При наближенні до об'єкту в певний момент камера наближається повільніше. Якщо вам потрібно максимально близько наблизитися до землі, то іноді це буває дуже складно зробити. Швидше за все, в найближчих оновленнях скролінг пофиксил, або навчать їм користуватися, що теж добре. У Unity 5 є проблеми з мультиплеер. Але, якщо у вас прямі руки, то він налаштовується дуже просто. Наприклад, в The Forest деякі об'єкти бачив один гравець, а інший - ні, а хороший приклад - гра HeartStone.[8]

2 Unreal Engine

The Unreal Engine - це ігровий рушій, розроблений Epic Games, вперше представлений у грі шутер від першої особи 1998 року Unreal. Хоча спочатку розроблений для шутерів від першої особи, він з успіхом застосовується в багатьох інших жанрах, включаючи платформи, бойові ігри, MMORPG та інші RPG. Написаний на C ++, Unreal Engine відрізняється високим ступенем портативності, підтримуючи широкий спектр платформ.[9]

Останній реліз - Unreal Engine 4, який запустився в 2014 році за моделлю підписки. З 2015 року її можна завантажити безкоштовно, її вихідний код доступний на GitHub. Epic дозволяє використовувати його в комерційних продуктах на основі роялті, зазвичай просять розробників 5% доходу від продажів.

Розвиток Unreal Engine 4 розпочався в 2003 році, за словами Марка Рейна, віце-президента Epic Games. Команда розшириться з часом, проектуючи рушій паралельно зусиллям команди UE3. У лютому 2012 року компанія Epic представила UE4 обмеженим учасникам на Конференції розробників ігор, а відео з рушієм, продемонстрованим технічним художником Аланом Віллардом, було оприлюднено публіці 7 червня 2012 року через GameTrailers TV.

Однією з найважливіших особливостей, запланованих для UE4, було глобальне освітлення в режимі реального часу за допомогою відстеження конусів вокселів, що виключало попередньо обчислене освітлення. Однак ця функція, що отримала назву Sparse Voxel Octree Global Illumination (SVOGI), була замінена аналогічним, але менш обчислювально дорогим алгоритмом через проблеми з продуктивністю. UE4 також включає нову систему візуальних сценаріїв "Креслення" (спадкоємця UE3 "Kismet"), яка дозволяє швидко розвивати логіку гри без використання коду, що призводить до меншої розриву між технічними художниками, дизайнерами та програмістами.

19 березня 2014 року на конференції розробників ігор (GDC) Epic Games випустили Unreal Engine 4 через нову ліцензійну модель. Для щомісячної підписки на рівні 19 доларів США розробникам було надано доступ до повної версії рушія, включаючи вихідний код C ++, який можна було завантажити через GitHub. Будь-який випущений продукт стягувався із 5% роялті валового доходу. Першою грою, випущеною за допомогою Unreal Engine 4, став Daylight, розроблений з раннім доступом до двигуна та випущений 29 квітня 2014 року.

4 вересня 2014 року Epic безкоштовно випустив Unreal Engine 4 до шкіл та університетів, включаючи особисті копії для студентів, які навчаються в акредитованих програмах розвитку відеоігор, інформатики, мистецтва, архітектури, моделювання та візуалізації. 19 лютого 2015 року Epic запустив Unreal Dev Grants - фонд розвитку на 5 мільйонів доларів, спрямований на надання грантів творчим проектам за допомогою Unreal Engine 4.

У березні 2015 року Epic випустив Unreal Engine 4 разом із усіма майбутніми оновленнями безкоштовно для всіх користувачів. В обмін на це Epic встановив вибірковий графік роялті, вимагаючи 5% доходу за продукцію, яка складає понад 3000 доларів на квартал.

Намагаючись залучити розробників Unreal Engine, Oculus VR в жовтні 2016 року оголосив, що сплачуватиме гонорари за всі титули Oculus Rift, оснащені Unreal, опубліковані в їхньому магазині до перших 5 мільйонів доларів валового доходу за гру.

Щоб підготуватися до випуску свого вільного режиму бойових роялей у Fortnite у вересні 2017 року, Epic довелося внести ряд модифікацій Unreal Engine, які допомогли йому обробити велику кількість (до 100) підключень до одного сервера зберігаючи високу пропускну здатність і покращуючи візуалізацію великого відкритого в грі світу. Епік заявив, що включить ці зміни до майбутніх оновлень Unreal Engine.

UnrealScript (часто скорочено до UScript) була рідною мовою сценаріїв Unreal Engine, яка використовувалася для створення коду ігор та ігрових подій до виходу Unreal Engine 4. Мова була розроблена для простого ігрового програмування високого рівня. Інтерпретатора UnrealScript запрограмував Sweeney, який також створив більш ранню мову сценаріїв гри, ZZT-oop.

Подібно до Java, UnrealScript був об'єктно-орієнтований без багатократного успадкування (класи всі успадковують із загального класу Object), а класи визначалися в окремих файлах, названих для класу, який вони визначають. На відміну від Java, UnrealScript не мав обгортки об'єктів для примітивних типів. Інтерфейси підтримувались лише в Unreal Engine покоління 3 та кількох іграх Unreal Engine 2. UnloadScript підтримується оператором перевантаження, але не методом перевантаження, за винятком додаткових параметрів.

На конференції розробників ігор 2012 року компанія Epic оголосила, що UnrealScript видаляється з Unreal Engine 4 на користь C ++. Візуальний сценарій підтримується системою Visual Scripting Blueprints, заміною попередньої системи візуального сценарію Kismet.[10]

2.4 Висновки до розділу 2:

- Визначено поняття ігрового рушія;
- Розглянуто аспекти розробки ігрових рушіїв;
- Проведений аналіз структури ігрових рушіїв, та його складники;
- Проведено аналіз сучасних потужних рушіїв: Unity і Unreal Engine;

3. РОЗРОБКА ГРАФІЧНОГО РУШІЯ

3.1 Розробка базового програмного забезпечення

Розробка базового програмного забезпечення для створення ігор на основі графічного sdk OpenGL використовуючи мову програмування Java та пакету LWJGL.

Мета даного розділу – створення графічного рушія, який може забезпечити відображення сцени гри в 3D, та розробка інтересу гравця для взаємодії з цими об’єктів. Як приклад використання та робото-спроможності рушія, буде створена RPG гра, в якій гравець зможе керувати персонажем, та вільно досліджувати світ гри.

Lightweight Java Game Library (LWJGL) - відкрита графічна бібліотека, основною метою якої є надання простого і легкого програмного інтерфейсу для творців комп'ютерних ігор на мові Java. LWJGL є високопродуктивної кроссплатформенной бібліотекою, яка широко використовується в розробці комп'ютерних ігор і мультимедійних додатках. Вона надає доступ до OpenGL, OpenAL, OpenCL і забезпечує платформонезавісимость доступ до різних маніпуляторам, таким як геймпади, керма та джойстики.[10]

Першим кроком для створення кожної гри є створення вікна відображення, для цього LWJGL надає програмістам ряд функцій на ряду з якими є метод Display(), за допомоги даного методу можна створити вікно графіки з певними налаштуваннями, для свого монітора я визначив висоту та ширину як 1280x1024 пікселі, та додав константу FPS_LIMIT що дорівнює 120, що обмежує використання ресурсів відеокарти для рендерінгу сцени, знизивши кількість запитів на побудову графіки до 120 разів на секунду.

Лістинг коду:

```
public class DisplayManager {
	private static final int WIDTH = 1280;
	private static final int HEIGHT = 1024;
	private static final int FPS_CAP = 120;
```

```
public static void createDisplay(){
	ContextAttribs attribs = new ContextAttribs(3,2)
	.withForwardCompatible(true)
	.withProfileCore(true);
	try {
		Display.setDisplayMode(new DisplayMode(WIDTH,HEIGHT));
		Display.create(new PixelFormat(), attribs);
		Display.setTitle("Test Engine");
	} catch (LWJGLException e) {
		e.printStackTrace();
	}
	GL11.glViewport(0,0, WIDTH, HEIGHT);
}
public static void updateDisplay(){
	Display.sync(FPS_CAP);
	Display.update();
}
public static void closeDisplay(){
	Display.destroy();
}
```
[11]

Наступним кроком є побудова графіки у вікні. Для цього існує два метод, найпростіший з них, власноруч створити масив точок, та у відповідності, прописати для методу рендерінгу, які з даних точок, будуть утворювати полігон щоб відобразити його у вікні. Такий спосіб підходить лише для примітивних моделей що будуть складатись з малої кількості полігонів. Для гри краще використати більш складний метод з використанням пам’яті відеокарти – використання списку масивів VBO та VAO.

Vertex Buffer Object (VBO) - це такий засіб OpenGL, що дозволяє завантажувати певні дані в пам'ять GPU. Наприклад, записати в GPU координати вершин, координати текстур або нормалі, потрібно створити VBO і покласти ці дані в нього. VBO буде створюється відповідно для кожного об’єктав в грі.

В свою чергу Vertex Arrays Object (VAO) - це такай масив, що говорить OpenGL, яку частину VBO слід використовувати в наступних командах. VAO являє собою масив, в елементах якої зберігається інформація про те, яку частину

якого VBO використовувати, і як ці дані потрібно інтерпретувати. Таким чином, один VAO за різними індексами може зберігати координати вершин, їх кольору, нормалі та інші дані. Переключившись на потрібний VAO можна ефективно звертатися до даних, на які він «вказує», використовуючи тільки індекси.[12]

Код, що реалізує використання даних списків:

```
private List<Integer> vaos = new ArrayList<Integer>();
private List<Integer> vbos = new ArrayList<Integer>();
private int createVAO(){
            int vaoID = GL30.glGenVertexArrays();
            vaos.add(vaoID);
            GL30.glBindVertexArray(vaoID);
            return vaoID;
      }
public RawModel loadToVAO(float[] positions,float[] textureCoords,float[] normals,int[] indices){
            int vaoID = createVAO();
            bindIndicesBuffer(indices);
            storeDataInAttributeList(0,3,positions);
            storeDataInAttributeList(1,2,textureCoords);
            storeDataInAttributeList(2,3,normals);
            unbindVAO();
            return new RawModel(vaoID,indices.length);
      }
private void storeDataInAttributeList(int attributeNumber, int coordinateSize,float[] data){
            int vboID = GL15.glGenBuffers();
            vbos.add(vboID);
            GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vboID);
            FloatBuffer buffer = storeDataInFloatBuffer(data);
            GL15.glBufferData(GL15.GL_ARRAY_BUFFER, buffer, GL15.GL_STATIC_DRAW);

GL20.glVertexAttribPointer(attributeNumber,coordinateSize,GL11.GL_FLOAT,false,0,0);
            GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, 0);
      }
```

[11]

Оскільки для даного рушія графічні об’єкти будуть використовуватись із сторонніх програм для створення 3D моделей, більшість сучасних 3д редакторів

підтримують формат «wavefront», тобто файли з розширенням .OBJ, даний формат є найкращим для використання разом зі списку масивів VBO та VAO. Оскілки данні про об’єкт в такому файлі записані послідовно, спочатку координати точок «v», координати накладання текстури «vt», вектори нормалі для кожної точки «vn», та інформацію про те, які данні необхідно використовувати, для побудування полігонів об’єкта «f», ці данні, з файлу можна одразу записувати в список VAO, в якому для кожного індексу VBO, представлена відповідна інформація координат, текстури, векторів нормалі, та відповідно інформацію точок полігону.

Код реалізації зчитування з файлу:

```
    public static ModelData loadOBJ(String objFileName) {
        FileReader isr = null;
        File objFile = new File(RES_LOC + objFileName + ".obj");
        try {
            isr = new FileReader(objFile);
        } catch (FileNotFoundException e) {
            System.err.println("File not found in res; don't use any extention");
        }
        BufferedReader reader = new BufferedReader(isr);
        String line;
        List<Vertex> vertices = new ArrayList<Vertex>();
        List<Vector2f> textures = new ArrayList<Vector2f>();
        List<Vector3f> normals = new ArrayList<Vector3f>();
        List<Integer> indices = new ArrayList<Integer>();
        try {
            while (true) {
                line = reader.readLine();
                if (line.startsWith("v ")) {
                    String[] currentLine = line.split(" ");
                    Vector3f vertex = new Vector3f((float)
Float.valueOf(currentLine[1]),
                            (float) Float.valueOf(currentLine[2]),
                            (float) Float.valueOf(currentLine[3]));

                    Vertex newVertex = new Vertex(vertices.size(), vertex);
                    vertices.add(newVertex);

                } else if (line.startsWith("vt ")) {
```

```
                String[] currentLine = line.split(" ");
                Vector2f texture = new Vector2f((float)
Float.valueOf(currentLine[1]),
                        (float) Float.valueOf(currentLine[2]));
                textures.add(texture);
            } else if (line.startsWith("vn ")) {
                String[] currentLine = line.split(" ");
                Vector3f normal = new Vector3f((float)
Float.valueOf(currentLine[1]),
                        (float) Float.valueOf(currentLine[2]),
                        (float) Float.valueOf(currentLine[3]));
                normals.add(normal);
            } else if (line.startsWith("f ")) {
                break;
            }
        }
        while (line != null && line.startsWith("f ")) {
            String[] currentLine = line.split(" ");
            String[] vertex1 = currentLine[1].split("/");
            String[] vertex2 = currentLine[2].split("/");
            String[] vertex3 = currentLine[3].split("/");
            processVertex(vertex1, vertices, indices);
            processVertex(vertex2, vertices, indices);
            processVertex(vertex3, vertices, indices);
            line = reader.readLine();
        }
        reader.close();
    } catch (IOException e) {
        System.err.println("Error reading the file");
    }
    removeUnusedVertices(vertices);
    float[] verticesArray = new float[vertices.size() * 3];
    float[] texturesArray = new float[vertices.size() * 2];
    float[] normalsArray = new float[vertices.size() * 3];
    float furthest = convertDataToArrays(vertices, textures, normals,
verticesArray,
            texturesArray, normalsArray);
    int[] indicesArray = convertIndicesListToArray(indices);
    ModelData data = new ModelData(verticesArray, texturesArray, normalsArray,
indicesArray,
            furthest);
    return data;
```

}[11]

Маючи інформацію про об’єкт, потрібно створити набір вказівок для відеокарти, як обробляти данні щоб відобразити об’єкт на сцені. Для цього використовуються GLSL (OpenGL Shading Language, Graphics Library Shader Language) - мова високого рівня для програмування шейдерів. Розроблено для виконання математики, яка зазвичай потрібна для виконання растеризації графіки. Синтаксис мови базується на мові програмування ANSI C, з якого були виключені багато можливостей, для спрощення мови і підвищення продуктивності. У мову включені додаткові функції і типи даних, наприклад для роботи з векторами і матрицями.

У відповідності до кожного параметру VBO передаємо обробку даних в GLSL. Тепер модель що зчитана з файлу може відображатись на екрані, проте даний об’єкт буде виглядати сплющеним в певну площину тому, що монітор комп’ютера - це 2D поверхня. 3D-сцена, відтворена OpenGL, повинна проектуватися на екран комп’ютера у вигляді 2D зображення.

Рисунок 3.1 – Відображення об’єкту на сцені

Для цього перетворення проекції використовується матриця GL_PROJECTION. Ця матриця перетворює всі вершинні дані від координат камери до координат сцени. Далі ці координати сцени також перетворюються на нормалізовані координати пристрою (NDC) шляхом ділення на w компонент координат сцени

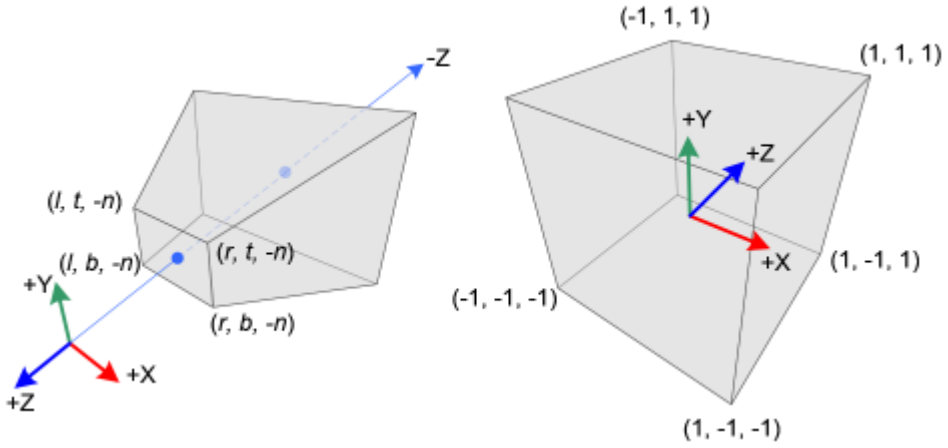

Рисунок 3.2 – Перетворення 3D координат сцени в стереоскопічне зображення.

У перспективному проектуванні 3D-точка в усіченій піраміді (координати очей) відображається на куб (NDC); діапазон координат x від [l, r] до [-1, 1], y-координата від [b, t] до [-1, 1] і z-координата від [-n, -f] до [-1, 1].

Координати очей визначаються в правій системі координат, але NDC використовує ліворучну систему координат. Тобто камера біля джерела дивиться вздовж осі Z у просторі очей, але вона дивиться вздовж осі Z у NDC.

У OpenGL 3D-точка в просторі очей проектується на найближчу площину (площину проекції). Наступні діаграми показують, як точка (xe, ye, ze) в просторі очей проектується на (xp, yp, zp) на найближчій площині.

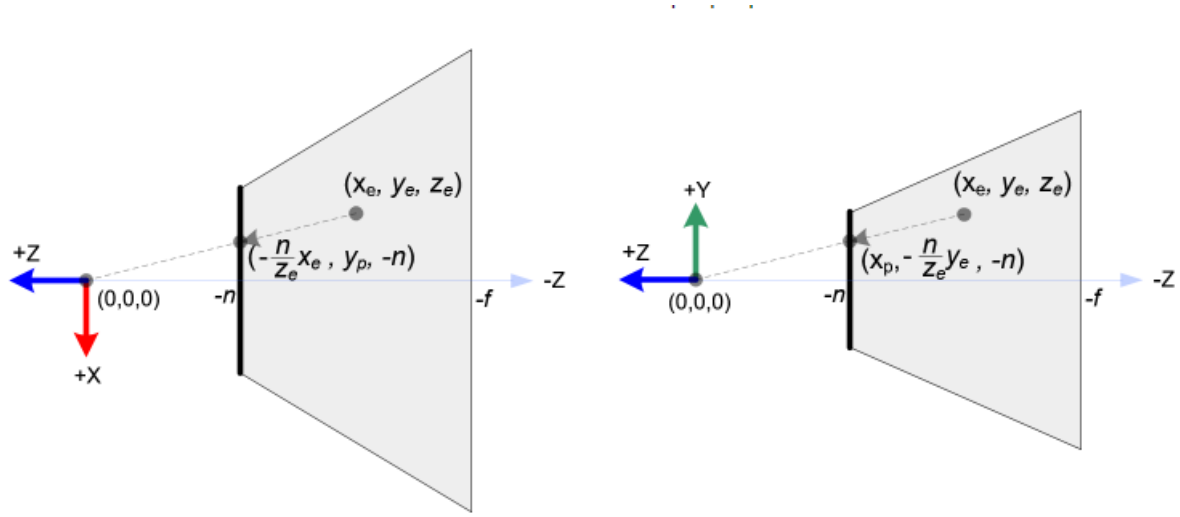

Рисунок 3.3 – Вигляд проекції геометрії з верху та з боку відповідно

Реалізувавши дану модель в через мову шейдерів, об’єкти на сцені тепер відображаються в проекції. Але поки данні об’єкти статичні, для реалізації динаміки потрібно доопрацювати код шейдерів.

В OpenGL камера статична і не може переміщуватись, вона завжди має координати (0,0) (як і більшості інших графічних бібліотек), тому в іграх при відображенні сцен, положення камери статична, а вся сцена рухається навколо камери, якщо реалізовувати зміну положення об’єктів тільки через таку матрицю, то положення об’єктів в сцені не можливо буд відслідкувати.

Щоб уникнути такий ефект потрібно створити матрицю яка буде опиратись на відносне положення об’єктив на сцені та тим, як користувач «змінює» положення камери, тобто об’єкти будуть мати свої статичні координати в сцені, але для камери на стороні відеокарти буде відбуватись математичне перетворення точок геометрії.[13]

Код матриці перетворення:

```
        public static Matrix4f createTransformationMatrix(Vector3f
translation, float rx, float ry,
                float rz, float scale) {
            Matrix4f matrix = new Matrix4f();
            matrix.setIdentity();
            Matrix4f.translate(translation, matrix, matrix);
            Matrix4f.rotate((float) Math.toRadians(rx), new Vector3f(1,0,0),
matrix, matrix);
```

```
                Matrix4f.rotate((float) Math.toRadians(ry), new Vector3f(0,1,0),
matrix, matrix);
                Matrix4f.rotate((float) Math.toRadians(rz), new Vector3f(0,0,1),
matrix, matrix);
                Matrix4f.scale(new Vector3f(scale,scale,scale), matrix, matrix);
                return matrix;
          }

          public static Matrix4f createViewMatrix(Camera camera) {
                Matrix4f viewMatrix = new Matrix4f();
                viewMatrix.setIdentity();
                Matrix4f.rotate((float)  Math.toRadians(camera.getPitch()),  new
Vector3f(1, 0, 0), viewMatrix,
                            viewMatrix);
                Matrix4f.rotate((float)   Math.toRadians(camera.getYaw()),   new
Vector3f(0, 1, 0), viewMatrix, viewMatrix);
                Vector3f cameraPos = camera.getPosition();
                Vector3f   negativeCameraPos   =   new   Vector3f(-cameraPos.x,-
cameraPos.y,-cameraPos.z);
                Matrix4f.translate(negativeCameraPos, viewMatrix, viewMatrix);
                return viewMatrix;
          }
```
[14]

Vertext-shader:

```
#version 400 core

in vec3 position;
in vec2 textureCoordinates;
in vec3 normal;

out vec2 pass_textureCoordinates;
void main(void){

      vec4 worldPosition = transformationMatrix * vec4(position,1.0);
      vec4 positionRelativeToCamera = viewMatrix * worldPosition;
      gl_Position = projectionMatrix * positionRelativeToCamera;
      pass_textureCoordinates = textureCoordinates;
}
Vertex shader:
#version 400 core

in vec2 pass_textureCoordinates;
```

```
in vec3 surfaceNormal;

out vec4 out_Color;

uniform sampler2D modelTexture;

void main(void){

	out_Color = vec4(diffuse,1.0) * texture (modelTexture, pass_textureCoordinates) + vec4(finalSpecular,1.0);
}
```

Для переміщення камери в сцені використовуються клавіші W,A,S,D, а для обертання камери використовується мишка.

```
public class Camera {

	private Vector3f position = new Vector3f(0,5,0);
	private Vector3f position = new Vector3f(0,5,0);
	private float pitch = 10;
	private float yaw ;
	public void move(){
		if(Keyboard.isKeyDown(Keyboard.KEY_P)){
			cameraMode = 1;
		}
		if(Keyboard.isKeyDown(Keyboard.KEY_O)){
			cameraMode = 0;
		}
		if (cameraMode == 0) {
			if(Mouse.isButtonDown(1)) {
				yaw -= Mouse.getDX() * 0.1f;
				pitch += Mouse.getDY() * 0.1f;
				pitch = Math.max(-90, Math.min(pitch, 90));
			}
			if (Keyboard.isKeyDown(Keyboard.KEY_W)) {
				float dx = (float)(Math.sin(Math.toRadians(yaw)));
				float dz = (float)(Math.cos(Math.toRadians(yaw)));
				float dy = (float)(Math.sin(Math.toRadians(pitch)));
				position.x +=dx;
				position.z -=dz;
				position.y -=dy;
```

```
                }
                if (Keyboard.isKeyDown(Keyboard.KEY_S)) {
                      float dx = (float)(Math.sin(Math.toRadians(yaw)));
                      float dz = (float)(Math.cos(Math.toRadians(yaw)));
                      float dy = (float)(Math.sin(Math.toRadians(pitch)));
                      position.x -=dx;
                      position.z +=dz;
                      position.y +=dy;
                }
                if (Keyboard.isKeyDown(Keyboard.KEY_D)) {
                      float dx = (float)(Math.sin(Math.toRadians(yaw-90)));
                      float dz = (float)(Math.cos(Math.toRadians(yaw-90)));
                      position.x -=dx;
                      position.z +=dz;
                }
                if (Keyboard.isKeyDown(Keyboard.KEY_A)) {
                      float dx = (float)(Math.sin(Math.toRadians(yaw+90)));
                      float dz = (float)(Math.cos(Math.toRadians(yaw+90)));
                      position.x -=dx;
                      position.z +=dz;
                }
          }
    }

    public Vector3f getPosition() {
          return position;
    }

    public float getPitch() {
          return pitch;
    }

    public float getYaw() {
          return yaw;
    }
}
```

3.2 Додавання світла

Наступним кроком у створенні рушія є додавання світла, щоб об’єкт змінював свою освітленість в залежності від нахилу і положення у відношенні до

точки світла. Дифузне відбиття побудовано так, що поверхні, які перпендикулярно спрямовані до джерела світла, виглядають яскравіше, ніж поверхні, де світло надходить під більш непрямим кутом. Ці об'єкти отримують більшу освітленість.

Для цього потрібно провести вектори від кожної точки до точки світла (тобто в зворотному напрямку).

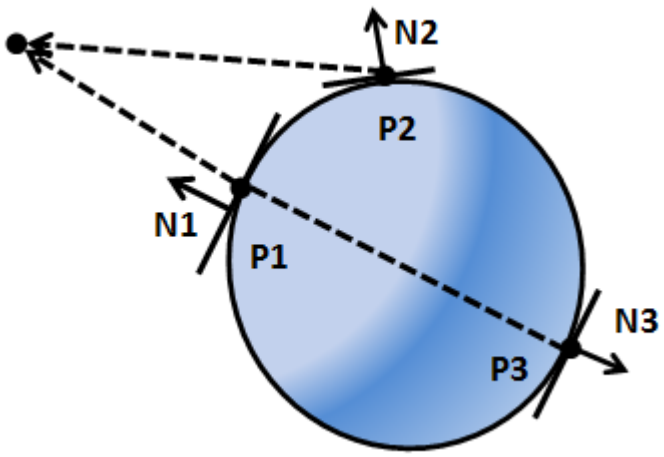

Рисунок 3.4 – Проекція світла на об’єкт

З малюнку видно, що нормаль пов'язана з точкою P1, названа N1 - паралельна вектору, який вказує на джерело світла. P1 має кут, рівний 0 з вектором, який вказує на джерело світла. Оскільки її поверхня перпендикулярна джерелу світла то P1 буде найяскравішою точкою на поверхні кулі.

Нормаль, пов'язана з точкою P2, названа N2, має кут близько 30 градусів з вектором, який вказує на джерело світла, тому він повинен мати менше освітлення ніж P1. А нормаль, пов'язана з точкою P3, і названа N3, також паралельна вектору, який вказує на джерело світла, але обидва вектора знаходяться у зворотному напрямку.

P3 має кут 180 градусів з вектором, який вказує на джерело світла, і взагалі не повинен отримувати світла. ?

На основі цього можна використати математичну операцію, яка називається крапковим продуктом. Ця операція займає два вектори і виробляє число (скаляр),

тобто позитивне, якщо кут між ними невеликий, і дає негативне число, якщо кут між ними широкий. Якщо обидва вектори нормалізуються, тобто обидва мають довжину, рівну одиниці, точковий добуток між ними буде в межах −1 і 1.

На основі цих даних створюймо шейдер, що буде відповідати за освітленість об’єктів в залежно від розміщення точкового світла.

Vertex shader

```
#version 400 core

in vec3 position;
in vec2 textureCoordinates;
in vec3 normal;

out vec2 pass_textureCoordinates;
out vec3 surfaceNormal;
out vec3 toLightVector;
out vec3 toCameraVector;
out float visibility;

uniform mat4 transformationMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform vec3 lightPosition;


void main(void){

      vec4 worldPosition = transformationMatrix * vec4(position,1.0);
      vec4 positionRelativeToCamera = viewMatrix * worldPosition;
      gl_Position = projectionMatrix * positionRelativeToCamera;
      pass_textureCoordinates = textureCoordinates;

      surfaceNormal = (transformationMatrix * vec4(actualNormal,0.0)).xyz;
      toLightVector = lightPosition - worldPosition.xyz;
      toCameraVector = (inverse(viewMatrix) * vec4(0.0,0.0,0.0,1.0)).xyz -
worldPosition.xyz;

}
```

[15]

Fragment shader:

```
#version 400 core

in vec2 pass_textureCoordinates;
in vec3 surfaceNormal;
in vec3 toLightVector;
in vec3 toCameraVector;

out vec4 out_Color;

uniform sampler2D modelTexture;
uniform vec3 lightColour;
uniform float shineDamper;
uniform float reflectivity;

void main(void){

	vec3 unitNormal = normalize(surfaceNormal);
	vec3 unitLightVector = normalize(toLightVector);

	float nDotl = dot(unitNormal,unitLightVector);
	float brightness = max(nDotl,0.2);
	vec3 diffuse = brightness * lightColour;

	vec3 unitVectorToCamera = normalize(toCameraVector);
	vec3 lightDirection = -unitLightVector;
	vec3 reflectedLightDirection = reflect(lightDirection,unitNormal);

	float specularFactor = dot(reflectedLightDirection , unitVectorToCamera);
	specularFactor = max(specularFactor,0.0);
	float dampedFactor = pow(specularFactor,shineDamper);
	vec3 finalSpecular = dampedFactor * reflectivity * lightColour;

	vec4 textureColour = texture(modelTexture,pass_textureCoordinates);
	if(textureColour.a< 0.5){
	    discard;
	}
```

```
            out_Color           =                       vec4(diffuse,1.0)           *
texture(modelTexture,pass_textureCoordinates) + vec4(finalSpecular,1.0);
```
}[16]

Тепер об’єкти, що знаходяться на сцені можуть змінювати своє положення, обертатись навколо осі, та змінювати свій розмір, і разом з цим буде змінюватись освітлення даного об’єкту. Приклад відображення об’єкту в сцені зображено на рис. 3.5.

Рисунок 3.5 – Поточний вигляд рендерінгу об’єкта на сцені

3.3 Створення карти розміщення об'єктів

Коли об’єкти готові, потрібно створити карту, на якій вони будуть розміщені та по якій буде пересуватись гравець.

Дана карта повинна підтримувати накладання текстур, та можливість використання карт висот. Тому, як об’єкт, карта складається з масиву полігонів, що мають певний заданий розмір, а їх кількість залежить від розширення карти

висот, яка накладається на данні полігони. Сама карта висот являє собою чорно-білу картинку в якій, кожному пікселю відповідає певний полігон, и чим світліший колір пікселя тим вище буде знаходитись дана точка на карті.

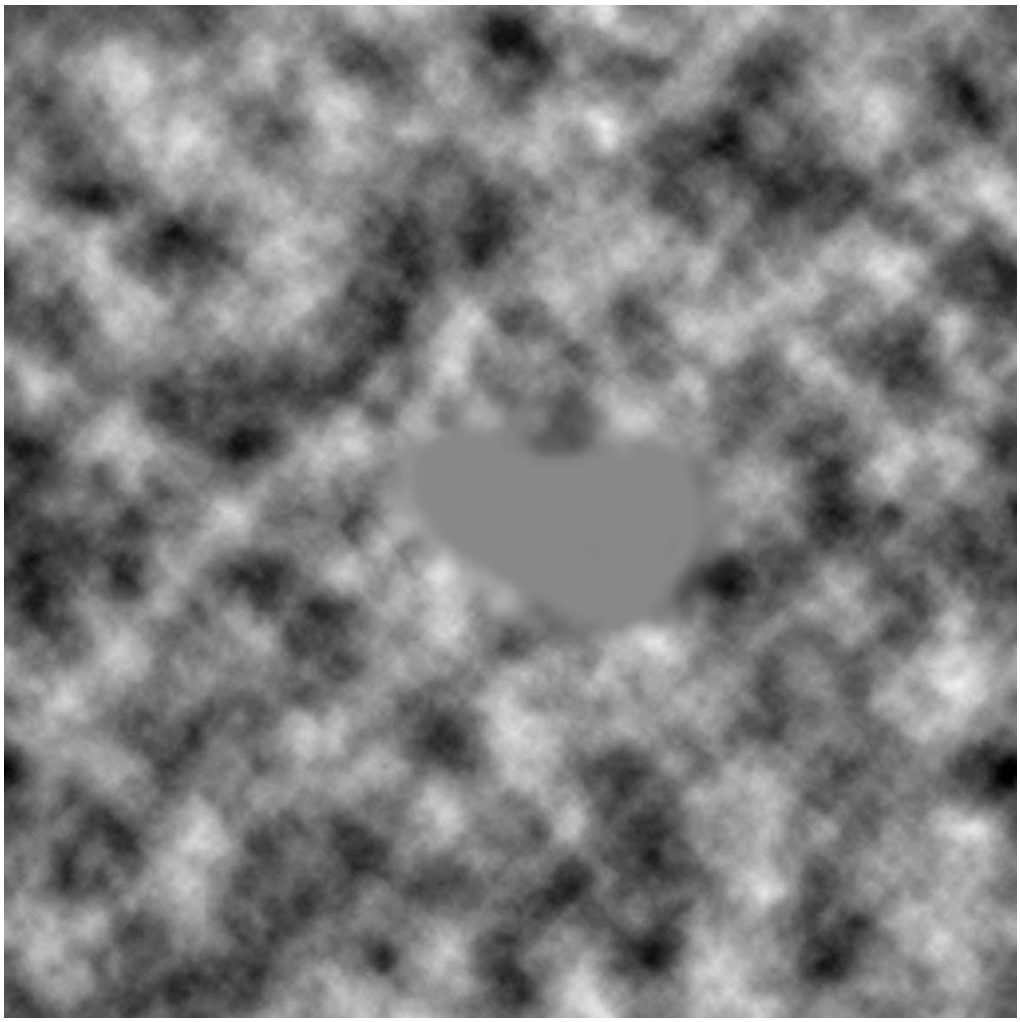

Рисунок 3.6 – приклад карти висот, що використовується.

Для накладання текстур, на поверхню карти, можна використати величезну текстуру карти, але така карта буде багато важити, а її рендерінг буде повільним і буде витрачати зайві ресурси відеокарти. Тому доцільно використати безшовні текстурі малого розміру, що повторюються на карті при її відображенні.

Однак однорідна текстура не може мати шляхи, і виглядає просто і не реалістичною. Тому в рушій потрібно додати підтримку накладання текстур. Найкращим чином це можна зробити через використання карти текстур, за аналогією роботи, вона подібна до карти висот, проте замість висоти, вона вказує

інформацію про те, яка частина карти, повинна мати ту або іншу текстуру. На рис. 3.7 зображена карта текстур.

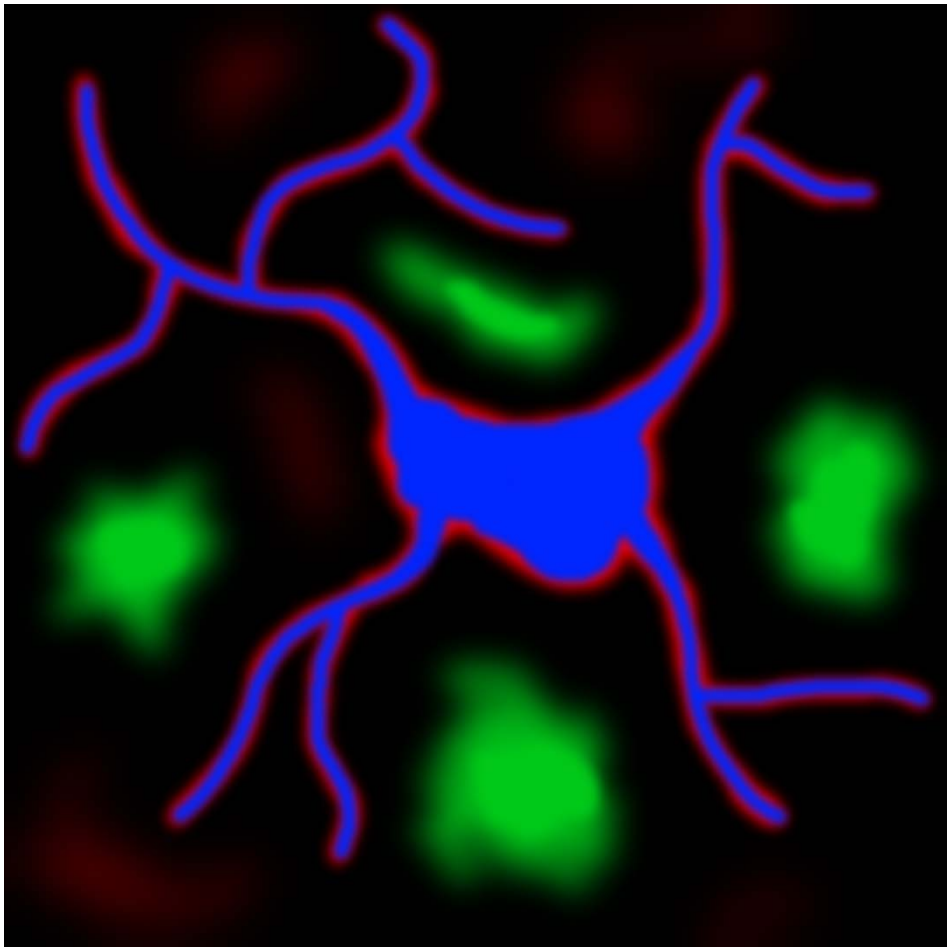

Рисунок 3.7 – Зображення карти висот

На даній карті кожному кольору виповідає певна текстура. Синьому кольору відповідає текстура кам’яної стежки, червоному виповідає притоптана трава, для того щоб згладити перехід від стежки до трави, зеленому кольору відповідає темніша текстура трави, а чорний виповідає за саму текстуру трави. На рис. 3.7 предаставленні текстури що використовуються при накладанні на карту.

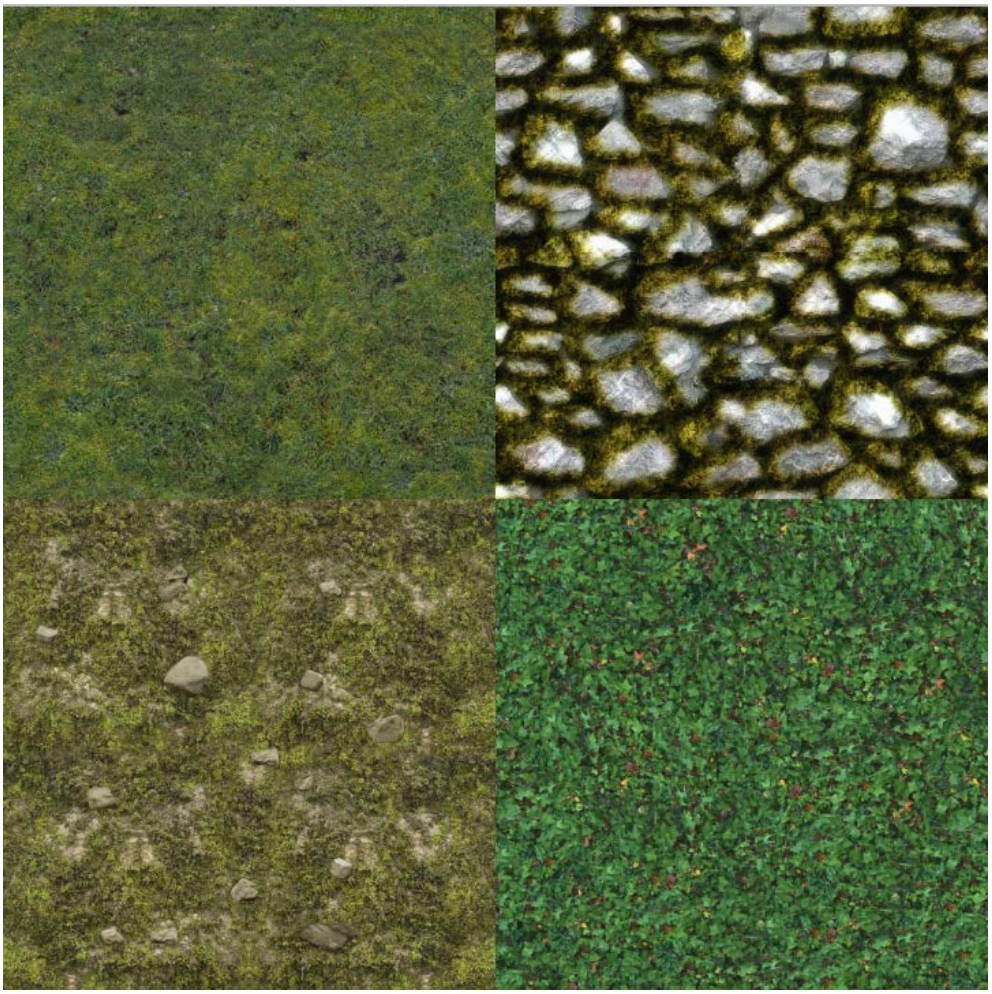

Рисунок 3.8 – Перелік текстур

Приклад реалізації коду:

```
public class Terrain {

	private static final float SIZE = 2000;
	private static final float MAX_HEIGHT = 25;
	private static final float MAX_PIXEL_COLOUR = 256*256*256;

	private float x;
	private float z;
	private RawModel model;
	private TerrainTexturePack texturePack;
	private TerrainTexture blendMap;

	private float[][] heights;

	public Terrain(int gridX, int gridZ, Loader loader, TerrainTexturePack
texturePack,TerrainTexture blendMap, String heightMap){
```

```
		this.texturePack = texturePack;
		this.blendMap = blendMap;
		this.x = gridX * SIZE + 1000;
		this.z = gridZ * SIZE + 1000;
		this.model = generateTerrain(loader, heightMap);
	}



	public float getX() {
		return x;
	}

	public float getZ() {
		return z;
	}

	public RawModel getModel() {
		return model;
	}

	public TerrainTexturePack getTexturePack() {
		return texturePack;
	}

	public TerrainTexture getBlendMap() {
		return blendMap;
	}

	public float getHeightOfTerrain(float worldX, float worldZ){
		float terrainX = worldX - this.x;
		float terrainZ = worldZ - this.z;
		float gridSquareSize = SIZE / ((float) heights.length - 1);
		int gridX = (int)Math.floor(terrainX/gridSquareSize);
		int gridZ = (int)Math.floor(terrainZ/gridSquareSize);
		if(gridX >= heights.length-1 ||gridZ >= heights.length-1 || gridX <
0|| gridZ < 0){
			return 0;
		}
		float xCoord = (terrainX % gridSquareSize)/ gridSquareSize;
		float zCoord = (terrainZ % gridSquareSize)/ gridSquareSize;
		float answer;
```

```
            if (xCoord <= (1-zCoord)) {
                  answer        =        Maths.barryCentric(new        Vector3f(0,
heights[gridX][gridZ], 0), new Vector3f(1,
                                        heights[gridX   +   1][gridZ],   0),   new
Vector3f(0,
                                        heights[gridX][gridZ   +   1],   1),   new
Vector2f(xCoord, zCoord));
            } else {
                  answer  =  Maths.barryCentric(new  Vector3f(1,  heights[gridX  +
1][gridZ], 0), new Vector3f(1,
                                        heights[gridX  +  1][gridZ  +  1],  1),  new
Vector3f(0,
                                        heights[gridX][gridZ   +   1],   1),   new
Vector2f(xCoord, zCoord));
            }
            return answer;
      }

      private RawModel generateTerrain(Loader loader,String heightMap){
            BufferedImage image = null;
            try {
                  image = ImageIO.read(new File("res/"+heightMap +".png") );
            } catch (IOException e) {
                  e.printStackTrace();
            }

            int VERTEX_COUNT = image.getHeight();

            heights = new float[VERTEX_COUNT][VERTEX_COUNT];

            int count = VERTEX_COUNT * VERTEX_COUNT;
            float[] vertices = new float[count * 3];
            float[] normals = new float[count * 3];
            float[] textureCoords = new float[count*2];
            int[] indices = new int[6*(VERTEX_COUNT-1)*(VERTEX_COUNT-1)];
            int vertexPointer = 0;
            for(int i=0;i<VERTEX_COUNT;i++){
                  for(int j=0;j<VERTEX_COUNT;j++){
                        vertices[vertexPointer*3] = (float)j/((float)VERTEX_COUNT -
1) * SIZE;
                        float height  = getHeight(j,i, image);
                        heights[j][i] = height;
```

```
                        vertices[vertexPointer*3+1] = height;
                        vertices[vertexPointer*3+2] = (float)i/((float)VERTEX_COUNT
- 1) * SIZE;
                        Vector3f normal = calculateNormal(j, i, image);
                        normals[vertexPointer*3] = normal.x;
                        normals[vertexPointer*3+1] = normal.y;
                        normals[vertexPointer*3+2] = normal.z;
                        textureCoords[vertexPointer*2]                          =
(float)j/((float)VERTEX_COUNT - 1);
                        textureCoords[vertexPointer*2+1]                        =
(float)i/((float)VERTEX_COUNT - 1);
                        vertexPointer++;
                  }
            }
            int pointer = 0;
            for(int gz=0;gz<VERTEX_COUNT-1;gz++){
                  for(int gx=0;gx<VERTEX_COUNT-1;gx++){
                        int topLeft = (gz*VERTEX_COUNT)+gx;
                        int topRight = topLeft + 1;
                        int bottomLeft = ((gz+1)*VERTEX_COUNT)+gx;
                        int bottomRight = bottomLeft + 1;
                        indices[pointer++] = topLeft;
                        indices[pointer++] = bottomLeft;
                        indices[pointer++] = topRight;
                        indices[pointer++] = topRight;
                        indices[pointer++] = bottomLeft;
                        indices[pointer++] = bottomRight;
                  }
            }
            return loader.loadToVAO(vertices, textureCoords, normals, indices);
      }

      private Vector3f calculateNormal(int x, int z, BufferedImage image){
            float heightL = getHeight(x-1,z,image);
            float heightR = getHeight(x+1,z,image);
            float heightD = getHeight(x,z-1,image);
            float heightU = getHeight(x,z+1,image);
            Vector3f  normal  =  new  Vector3f(  heightL  -  heightR,  2f,  heightD  -
heightU);
            normal.normalise();
            return normal;
```

```
        }

}
```

Після накладання текстур і карти висот, отриманий наступний результат:

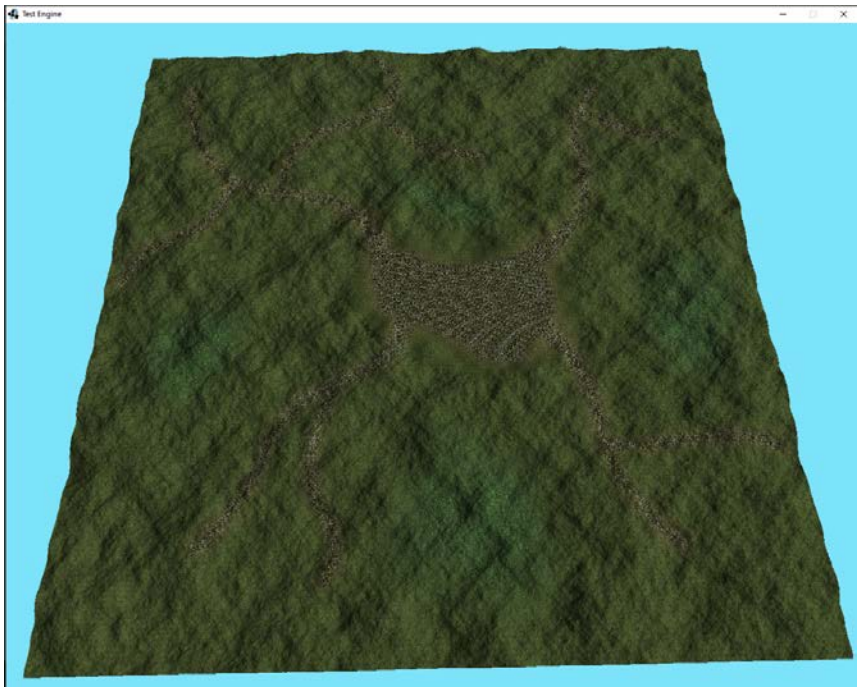

Рисунок 3.9 – Зображення карти з висоти птичого польоту

Для перспективи від першої особи дана карта має наступний вигляд:

Рисунок 3.10 – Зображення карти від першої особи

3.4 Створення ігрового персонажу

Фінальним кроком є створення ігрового персонажу, що буде переміщуватись по карті.

Такий об’єкт гравця можна створити наслідувавши параметри і методи від класу об’єкта, надавши йому переміщення, через ввід користувача. На стрілки вліво та вправо гравець зможе обертати ігрового персонажа, на стрілки вперед та назад, персонаж буде пересуватись у напрямку, в який він дивиться, та відповідно назад від нього. А на кнопку О персонаж буде виконувати стрибок.

Реалізація коду:

```
public class Player extends Entity{

    private static final float RUN_SPEED = 10;
    private static final float TURN_SPEED = 200;
    private static final float GRAVITY = -50;
    private static final float JUMP_POWER = 20;

    public static final float TERRAIN_HEIGHT = 0;

    private float currentSpeed= 0;
    private float currentTurnSpeed = 0;
    private float upwardsSpeed = 0;


    public Player(TexturedModel model, Vector3f position, float rotX, float
rotY, float rotZ, float scale) {
        super(model, position, rotX, rotY, rotZ, scale);
    }
    public void move(){
        checkInput();
        super.increaseRotation(0,currentTurnSpeed*
DisplayManager.getFrameTimeSeconds(),0);
        float distance = currentSpeed* DisplayManager.getFrameTimeSeconds();
        float        dx                =        (float)(distance        *
Math.sin(Math.toRadians(super.getRotY())));
```

```
            float dz = (float)(distance *
Math.cos(Math.toRadians(super.getRotY())));
            super.increasePosition(dx,0,dz);
            upwardsSpeed += GRAVITY * DisplayManager.getFrameTimeSeconds();
            super.increasePosition(0,upwardsSpeed *
DisplayManager.getFrameTimeSeconds(),0);
            if(super.getPosition().y < TERRAIN_HEIGHT){
                upwardsSpeed = 0;
                super.getPosition().y = TERRAIN_HEIGHT;
            }
        }


        private void checkInput(Terrain terrain){
            if(Keyboard.isKeyDown(Keyboard.KEY_UP)){
               this.currentSpeed= RUN_SPEED;
            }else if(Keyboard.isKeyDown(Keyboard.KEY_DOWN)){
                this.currentSpeed= -RUN_SPEED;
            }else {
                this.currentSpeed = 0;
            }
            if(Keyboard.isKeyDown(Keyboard.KEY_LEFT)){
                this.currentTurnSpeed= TURN_SPEED;
            } else if(Keyboard.isKeyDown(Keyboard.KEY_RIGHT)){
                this.currentTurnSpeed= -TURN_SPEED;
            }else {
                this.currentTurnSpeed = 0;
            }
            if(Keyboard.isKeyDown(Keyboard.KEY_J)){
                if(super.getPosition().y == TERRAIN_HEIGHT) {
                    jump();
                }
            }
        }
    }
```

Модель персонажа зображена на рис. 3.9

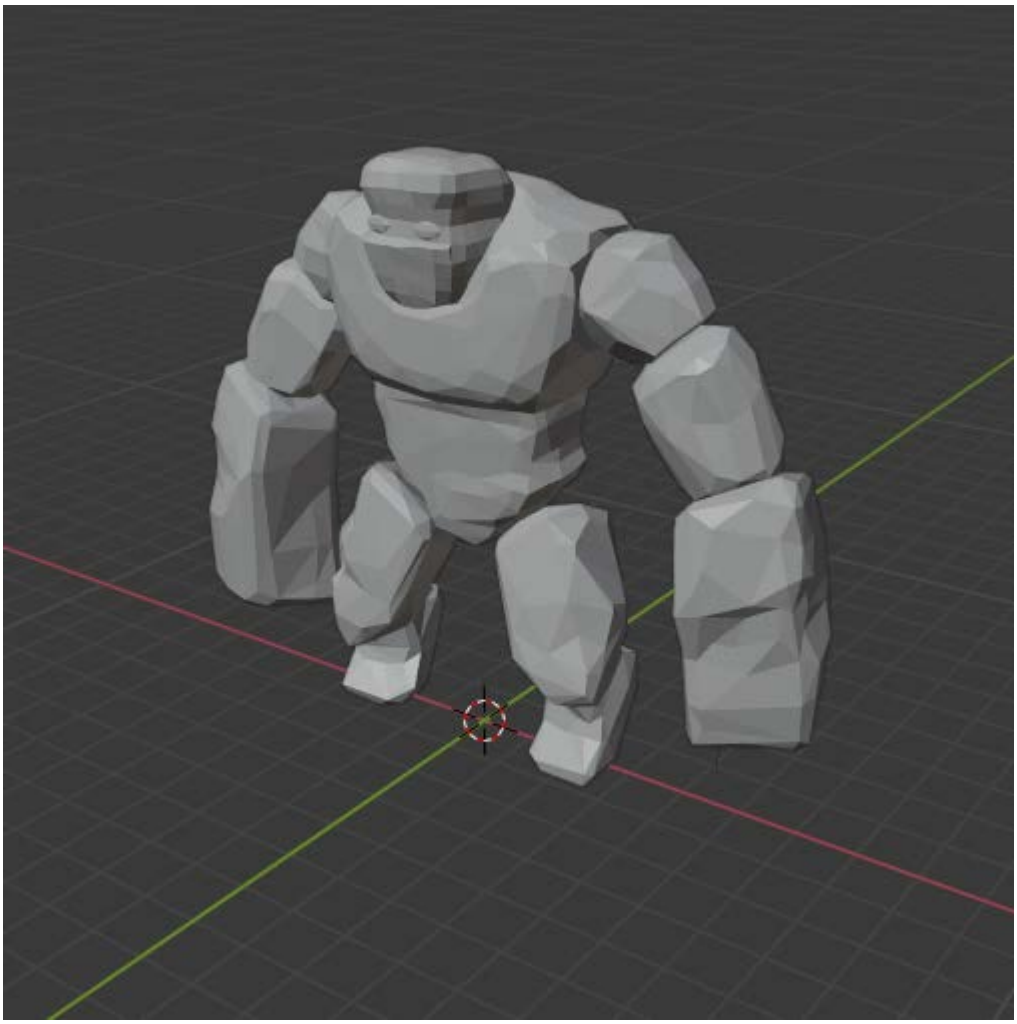

Рисунок 3.11 – Ігровий персонаж без текстури

Оскільки карта по якій ходить персонаж не рівномірна, то і його модель повинна знаходиться на поверхні карти, тобто мати колізію з нею. Щоб створити таку колізію, необхідно постійно зчитувати координати положення персонажа в світі, за цими координатами знаходити полігон, на якому стоїть персонаж, і через середньоквадратичне значення висот точок, що формують полігон, повертати це значення персонажу, як мінімальний рівень його моделі.

Код що реалізує дану функцію:

```
private float getHeight(int x, int z, BufferedImage image){
            if(x<0    ||    x>=    image.getHeight()    ||    z<0    ||    z>=
image.getHeight()){
                  return 0;
            }
            float height = image.getRGB(x,z);
            height += MAX_PIXEL_COLOUR/2f;
            height /=MAX_PIXEL_COLOUR/2f;
```

```
        height *= MAX_HEIGHT;
        return height;
    }
```

Повна сцена гри має тепер наступний вигляд:

Рисунок 3.12 Сцена гри с персонажем, та ігровими об’єктами

В створеному ігровому рушії, розробник ігор, може створити власного ігрового персонажа що буде вільно пересуватись по світу, створити для гри карту, та наповнити цей світ різноманітними об’єктами, що відповідає жанру пригодницьких РПГ ігор, на що і розрахований даний ігровий рушій.

3.5 Висновки до розділу 3:

- Було розглянуто графічну бібліотеку OpenGL, та пакет бібліотек JLWGL;

- На базі графічної бібліотеки, був розроблений ігровий рушій, який дозволяє створювати 3D ігри жанру 3rd-person advanture. Ігри створені на базі розробленого рушія підтримують створення ігрового персонажа, який може вільно пересуватись по ігровій карті, ігрову карту, що підтримує карту висот, та накладання текстур, камеру що можна пересувати по ігровому світу, просте освітлення, та завантаження моделей з wavefront файлів;

- Запропоновано, в продовження розробки ігрового рушія, підтримку використання фізичних бібліотек, анімації та звуку, і доопрацювання поточних можливостей рушія для створення повноцінних ігор;

- На основі роботи було створено кілька комп’ютерних практикумів для вивчення дисципліни «Технології створення освітніх комп'ютерних ігор та проектування доповненої реальності».

ВИСНОВКИ

У ході виконання даної дипломної роботи було досліджено:

1. Поняття комп’ютерної гри, та основні етапи історії розвитку ігрової індустрії. Основні жанри ігор та їх особливості, і механіки.

2. Поняття ігрового рушія, його основних складових, таких, як: графічна складова, фізична складова, анімаційна, і аудіо складові, інструменти відладки, елементи вводу-виводу, та користувальницький інтерфейс.

3. Також, було розроблене, базове графічне програмне забезпечення, з використанням бібліотек OpenGl та lwjgl. Був написаний код, що реалізує відображення 3D об’єктів в сцені з різними параметрами трансформації, ігрову карту, що підтримує накладання текстур і використання карт висот, та ігрового персонажа що може вільно переміщуватись по карті.

4. Даний ігровий рушій можна поступово покращувати, і надавати нових можливостей, втім за допомоги даного коду можна реалізувати ігри інших жанрів (наприклад гонки, якщо замінити модель персонажа на модель автомобіля), або далі розвивати в напрямку RPG ігор з видом від третьої особи.

5. Даний рушій був запропонований, як база для створення практикум з дисципліни «Технології створення освітніх комп’ютерних ігор та проектування доповненої реальності» для вивчення базових складових ігрових рушіїв, та їх застосування для створення ігор.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. В.В. Макаренко, К.О. Трапезон, А.М. Чермянін. Основні вимоги до оформлення атестаційних робіт, дипломних та курсових проектів: методичні рекомендації для студентів усіх форм навчання факультету електроніки. – К.: ФЕЛ НТУУ “КПІ”, 2006. – 112 с.

2. Video game URL: https://en.wikipedia.org/wiki/Video_game (дата звернення: 23.05.2020).

3. История компьютерных игр URL: https://sites.google.com/site/historygamesabc/istoria-komputernyh-igr (дата звернення: 20.05.2020).

4. История развития и эволюция видеоигр в цифрах и картинках URL: https://hype.tech/@id103/istoriya-razvitiya-i-evolyuciya-videoigr-v-cifrah-i-kartinkah-w4e9feon (дата звернення: 23.05.2020).

5. История развития компьютерных игр - от первых игр до виртуальной реальности URL:http://stevsky.ru/starie-igri/istoriya-razvitiya-igr-ot-pervich-igr-do-virtualnoy-realnosti (дата звернення: 22.05.2020).

6. Jason Gregory. Game Engine Architecture. A K Peters, Ltd. Wellesley, Massachusetts, 2009. 853 p.

7. Игровой движок URL: https://ru.wikipedia.org/wiki/Игровой_движок (дата звернення: 22.05.2020).

8. Полный обзор Unity 5 URL: http://devgam.com/polnyj-obzor-unity-5 (дата звернення: 23.05.2020).

9. Unreal Engine URL: https://en.wikipedia.org/wiki/Unreal_Engine#Unreal_Engine_5 (дата звернення: 17.05.2020).

10. Lightweight Java Game Library URL: https://ru.wikipedia.org/wiki/Lightweight_Java_Game_Library (дата звернення: 26.05.2020).

11. Examples Java engine code realization URL: https://pastebin.com/e9V2qP5u (дата звернення: 25.05.2020).

12. Изучение OpenGL: VBO, VAO и шейдеры URL https://eax.me/opengl-vbo-vao-shaders/ (дата звернення: 14.05.2020).

13. OpenGL Projection Matrix URL: http://www.songho.ca/opengl/gl_projectionmatrix.html (дата звернення: 25.05.2020).

14. Matrix transformation java OpenGL URL: https://www.java-forum.org/thema/meine-funktion-um-die-hoehe-des-terrains-an-bestimmter-position-in-diesem-fall-spieler-position-zu-bekommen-giebt-nur-0-zurueck.185860/ (дата звернення: 18.05.2020).

15. Terrain code examples for jlwgl URL: https://gamedev.stackexchange.com/questions/174176/flat-shading-does-not-work-correctly-opengl (дата звернення: 21.05.2020).

ДОДАТОК А
«SUMMARY»

Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop both three-dimensional and two-dimensional video games andsimulations for computers, consoles, and mobile devices. First announced only for OS X at Apple's Worldwide Developers Conference in 2005, it has since been extended to target 27 platforms. Six major versions of Unity have been released. For a list of games made with Unity, visit List of Unity games.

Unity is a multipurpose game engine that supports 2D and 3D graphics, drag-and-drop functionality and scripting using C#. Two other programming languages were supported:Boo, which was deprecated with the release of Unity 5 and JavaScript which started its deprecation process in August 2017 after the release of Unity 2017.1.
The engine targets the following graphics APIs: Direct3D on Windows and Xbox One; OpenGL on Linux, macOS, and Windows; OpenGL ES on Android and iOS; WebGL on the web; and proprietary APIs on the video game consoles. Additionally, Unity supports the low-level APIs Metal on iOS and macOS and Vulkan on Android, Linux, and Windows, as well as Direct3D 12 on Windows and Xbox One.

Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.[7] Unity also offers services to developers, these are: Unity Ads, Unity Analytics, Unity Certification, Unity Cloud Build, Unity Everyplay, Unity IAP, Unity Multiplayer, Unity Performance Reporting and Unity Collaborate.

Unity supports the creation of custom vertex, fragment (or pixel), tesselation, compute shaders and Unity's own surface shaders using Cg, a modified version of Microsoft's High-Level Shading Language. Unity supports building to 27 different platforms.

The platforms are listed in the following: iOS, Android, Tizen, Windows, Universal WindowsPlatform, Mac, Linux, WebGL, PlayStation4, PlayStationVita, Xbox One, Wii U, 3DS, Oculus Rift, Google Cardboard, Steam VR, Play Station VR, Gear VR, Windows Mixed Reality, Daydream, AndroidTV, Samsung Smart TV, tvOS, Nintendo Switch, Fire OS, Facebook Gameroom, Apple ARKit, Google AR Core, and Vuforia.

Unity formerly supported seven other platforms including its own Unity Web Player. Unity Web Player was a browser plugin that was supported in Windows and OS X only,[1] which has been deprecated in favor of Web GL

Unity is the default software development kit (SDK) for Nintendo's Wii U video game console platform, with a free copy included by Nintendo with each Wii U developer license. Unity Technologies calls this bundling of a third-party SDK an "industry first".

In 2012, VentureBeat said, "Few companies have contributed as much to the flowing of independently produced games as Unity Technologies. ... More than 1.3 million developers are using its tools to create gee-whiz graphics in their iOS, Android, console, PC, and web-based games. ... Unity wants to be the engine for multi-platform games, period."

For the Apple Design Awards at the 2006 WWDC trade show, Apple, Inc. named Unity as the runner-up for its Best Use of Mac OS X Graphics category, a year after Unity's launch at the same trade show. Unity Technologies says this is the first time a game design tool has ever been nominated for this award. A May 2012 survey by *Game Developer* magazine indicated Unity as its top game engine for mobile platforms. In July 2014, Unity won the "Best Engine" award at the UK's annual Develop Industry Excellence Awards.

Unity 5 was met with similar praise, with *The Verge* stating that "Unity started with the goal of making game development universally accessible.... Unity 5 is a long-awaited step towards that future."

Following the release of Unity 5, Unity Technologies drew some criticism for the high volume of quickly produced games published on the Steam distribution

platform by inexperienced developers. CEO John Riccitiello said in an interview that he believes this to be a side-effect of Unity's success in democratizing game development: "If I had my way, I'd like to see 50 million people using Unity – although I don't think we're going to get there any time soon. I'd like to see high school and college kids using it, people outside the core industry. I think it's sad that most people are consumers of technology and not creators. The world's a better place when people know how to create, not just consume, and that's what we're trying to promote."

In December 2016, Unity Technologies announced that they will change the versioning numbering system for Unity from sequence-based identifiers to year of release to align the versioning with their more frequent release cadence.

On December 16, 2013, Unity Technologies Japan revealed new screenshots for an official mascot character named Unity-chan (ユニティちゃん *Yuniti-chan*), real name Kohaku Ōtori (大鳥 こはく *Ōtori Kohaku*) (voiced by Asuka Kakumoto), with exhibit of the character in Comic Market 85 event in the Tokyo Big Sight between December 29 to the 31st, where themed goods would be distributed and her voice actress would be present at the event. The character's associated game data were to be released in spring 2014. The character was designed by Unity Technologies Japan designer "ntny" as an open-source heroine character. The company allows the use of Unity-chan and related characters in secondary projects under certain licenses.[25] For example, Unity-chan appears as a playable character in *Runbow*. The popularity of the character also led to her appearance in VOCALOIDadaptions, including her own sound library for VOCALOID4 and a special adaption of VOCALOID designed to work with the Unity Engine 5.0 version called Unity with VOCALOID.[6]

ДОДАТОК В

Лістинг коду реалізаціїї графічного рушія

Код рушіяії реалізований на мові java, використовуючи бібліотеки lwjgl:

```
public class MainGameLoop {


      public static void main(String[] args) {


            System.setProperty("org.lwjgl.librarypath", new
File("lib/natives").getAbsolutePath());

            System.setProperty("org.lwjgl64.librarypath", new
File("lib/natives").getAbsolutePath());


            DisplayManager.createDisplay();

            Loader loader = new Loader();


            TerrainTexture backGroundTexture = new
TerrainTexture(loader.loadTexture("grass"));

            TerrainTexture rTexture = new
TerrainTexture(loader.loadTexture("moss"));

            TerrainTexture gTexture = new
TerrainTexture(loader.loadTexture("grass2"));

            TerrainTexture bTexture = new
TerrainTexture(loader.loadTexture("rock"));


            TerrainTexturePack texturePack  = new
TerrainTexturePack(backGroundTexture, rTexture,gTexture,bTexture);

            TerrainTexture blendMap = new
TerrainTexture(loader.loadTexture("map"));

            Terrain terrain = new Terrain(-1,-1,loader,texturePack,blendMap,
"heightmap");


            ModelData data = OBJFileLoader.loadOBJ("cottage_blender");

            RawModel cottage =
loader.loadToVAO(data.getVertices(),data.getTextureCoords(),data.getNormals(),data
.getIndices());

            ModelTexture cottage1 = new
ModelTexture(loader.loadTexture("house_tex"));

            TexturedModel staticModel = new TexturedModel(cottage,cottage1);




            ModelData model1 = OBJFileLoader.loadOBJ("grass1");
```

```
            RawModel grass =
loader.loadToVAO(model1.getVertices(),model1.getTextureCoords(),model1.getNormals(
),model1.getIndices());

            ModelTexture text = new ModelTexture(loader.loadTexture("grass1"));


            ModelData well = OBJFileLoader.loadOBJ("Well");

            RawModel well1 =
loader.loadToVAO(well.getVertices(),well.getTextureCoords(),well.getNormals(),well
.getIndices());

            ModelTexture well_tex = new
ModelTexture(loader.loadTexture("Well_tex"));


            text.setHasTransparency(true);

            text.setUseFakeLighting(true);


            TexturedModel staticModelwell = new TexturedModel(well1,well_tex);

            TexturedModel staticModel1 = new TexturedModel(grass,text);

            Entity wellEnt = new Entity(staticModelwell, new
Vector3f(6,terrain.getHeightOfTerrain(6,6),6),0,0,0,2f);


            RawModel robot = OBJLoader.loadObjModel("Stone",loader);

            TexturedModel texRobot = new TexturedModel(robot,new
ModelTexture(loader.loadTexture("Stone_tex")));

            Player player = new Player(texRobot,new Vector3f(0,0,0),0,0,0,0.68f);




            List<Entity> entities = new ArrayList<Entity>();

            Random random = new Random();

            for(int i=0;i<30;i++){

                  float x = random.nextFloat()*600 -200;

                  float z = random.nextFloat() * 400 -200;

                  float y = terrain.getHeightOfTerrain(x,z);

                  entities.add(new Entity(staticModel, new
Vector3f(x,y,z),0,random.nextFloat() *360,0,1f));

            }


            List<Entity> entities1 = new ArrayList<Entity>();

            for(int i=0;i<2000;i++){

                  float x = random.nextFloat()*80 - 40;
```

```
                float z = random.nextFloat()*80 - 300;
                float y = terrain.getHeightOfTerrain(x,z);
                entities1.add(new Entity(staticModel1, new
Vector3f(x,y,z),0,random.nextFloat() *360,0,1f));
            }



            Light light = new Light(new Vector3f(20000,20000,2000),new
Vector3f(1,1,1));




            Camera camera = new Camera(player);
            MasterRenderer renderer = new MasterRenderer();

            while(!Display.isCloseRequested()){
                  camera.move();
                  player.move(terrain);
                  renderer.processEntity(player);
                  renderer.processTerrain(terrain);
                  renderer.processEntity(wellEnt);
                  for(Entity entity:entities){
                        renderer.processEntity(entity);
                  }
                  for(Entity entity:entities1){
                        renderer.processEntity(entity);
                  }
                  renderer.render(light, camera);
                  DisplayManager.updateDisplay();
            }

            renderer.cleanUp();
            loader.cleanUp();
            DisplayManager.closeDisplay();
      }
}
```

[11]

```
public class Camera {

	private Vector3f position = new Vector3f(0,5,0);
	private float pitch = 10;
	private float yaw ;
	private float roll;

	private int cameraMode = 0;

	private Player player;

	public Camera(Player player){
		this.player=player;
	}

	public void move(){
		if(Keyboard.isKeyDown(Keyboard.KEY_P)){
			cameraMode = 1;
		}
		if(Keyboard.isKeyDown(Keyboard.KEY_O)){
			cameraMode = 0;
		}
		if (cameraMode == 0) {
			if(Mouse.isButtonDown(1)) {
				yaw -= Mouse.getDX() * 0.1f;
				pitch += Mouse.getDY() * 0.1f;
				pitch = Math.max(-90, Math.min(pitch, 90));
			}
			if (Keyboard.isKeyDown(Keyboard.KEY_W)) {
				float dx = (float)(Math.sin(Math.toRadians(yaw)));
				float dz = (float)(Math.cos(Math.toRadians(yaw)));
				float dy = (float)(Math.sin(Math.toRadians(pitch)));
				position.x +=dx;
				position.z -=dz;
				position.y -=dy;
```

```
            }
            if (Keyboard.isKeyDown(Keyboard.KEY_S)) {
                  float dx = (float)(Math.sin(Math.toRadians(yaw)));
                  float dz = (float)(Math.cos(Math.toRadians(yaw)));
                  float dy = (float)(Math.sin(Math.toRadians(pitch)));
                  position.x -=dx;
                  position.z +=dz;
                  position.y +=dy;
            }
            if (Keyboard.isKeyDown(Keyboard.KEY_D)) {
                  float dx = (float)(Math.sin(Math.toRadians(yaw-90)));
                  float dz = (float)(Math.cos(Math.toRadians(yaw-90)));
                  position.x -=dx;
                  position.z +=dz;
            }
            if (Keyboard.isKeyDown(Keyboard.KEY_A)) {
                  float dx = (float)(Math.sin(Math.toRadians(yaw+90)));
                  float dz = (float)(Math.cos(Math.toRadians(yaw+90)));
                  position.x -=dx;
                  position.z +=dz;
            }
            if (Keyboard.isKeyDown(Keyboard.KEY_SPACE)) {
                  position.y += 1f;
            }
            if (Keyboard.isKeyDown(Keyboard.KEY_LSHIFT)) {
                  position.y -= 1f;
            }
      }
}

public Vector3f getPosition() {
      return position;
}
```

```
	public float getPitch() {
		return pitch;
	}

	public float getYaw() {
		return yaw;
	}

	public float getRoll() {
		return roll;
	}

	public int getCameraMode() {
		return cameraMode;
	}

	public void setCameraMode(int cameraMode) {
		this.cameraMode = cameraMode;
	}

}
public class Entity {

	private TexturedModel model;
	private Vector3f position;
	private float rotX, rotY, rotZ;
	private float scale;

	public Entity(TexturedModel model, Vector3f position, float rotX, float
rotY, float rotZ,
			float scale) {
		this.model = model;
		this.position = position;
		this.rotX = rotX;
		this.rotY = rotY;
		this.rotZ = rotZ;
```

```
		this.scale = scale;
	}

	public void increasePosition(float dx, float dy, float dz) {
		this.position.x += dx;
		this.position.y += dy;
		this.position.z += dz;
	}

	public void increaseRotation(float dx, float dy, float dz) {
		this.rotX += dx;
		this.rotY += dy;
		this.rotZ += dz;
	}

	public TexturedModel getModel() {
		return model;
	}

	public void setModel(TexturedModel model) {
		this.model = model;
	}

	public Vector3f getPosition() {
		return position;
	}

	public void setPosition(Vector3f position) {
		this.position = position;
	}

	public float getRotX() {
		return rotX;
	}
```

```
	public void setRotX(float rotX) {
		this.rotX = rotX;
	}

	public float getRotY() {
		return rotY;
	}

	public void setRotY(float rotY) {
		this.rotY = rotY;
	}

	public float getRotZ() {
		return rotZ;
	}

	public void setRotZ(float rotZ) {
		this.rotZ = rotZ;
	}

	public float getScale() {
		return scale;
	}

	public void setScale(float scale) {
		this.scale = scale;
	}

}
public class Light {

	private Vector3f position;
	private Vector3f colour;

	public Light(Vector3f position, Vector3f colour) {
```

```
            this.position = position;
            this.colour = colour;
      }

      public Vector3f getPosition() {
            return position;
      }

      public void setPosition(Vector3f position) {
            this.position = position;
      }

      public Vector3f getColour() {
            return colour;
      }

      public void setColour(Vector3f colour) {
            this.colour = colour;
      }


}
```
[11]

```
public class Player extends Entity{

    private static final float RUN_SPEED = 10;
    private static final float TURN_SPEED = 200;
    private static final float GRAVITY = -50;
    private static final float JUMP_POWER = 20;

    public static final float TERRAIN_HEIGHT = 0;

    private float currentSpeed= 0;
    private float currentTurnSpeed = 0;
    private float upwardsSpeed = 0;
```

```
    public Player(TexturedModel model, Vector3f position, float rotX, float rotY,
float rotZ, float scale) {
        super(model, position, rotX, rotY, rotZ, scale);
    }
    public void move(Terrain terrain){
        checkInput(terrain);
        super.increaseRotation(0,currentTurnSpeed*
DisplayManager.getFrameTimeSeconds(),0);
        float distance = currentSpeed* DisplayManager.getFrameTimeSeconds();
        float dx  = (float)(distance * Math.sin(Math.toRadians(super.getRotY())));
        float dz  = (float)(distance * Math.cos(Math.toRadians(super.getRotY())));
        super.increasePosition(dx,0,dz);
        upwardsSpeed += GRAVITY * DisplayManager.getFrameTimeSeconds();
        super.increasePosition(0,upwardsSpeed *
DisplayManager.getFrameTimeSeconds(),0);
        float terrainHeight =
terrain.getHeightOfTerrain(super.getPosition().x,super.getPosition().z);
        if(super.getPosition().y < terrainHeight){
            upwardsSpeed = 0;
            super.getPosition().y = terrainHeight;
        }
    }



    private void checkInput(Terrain terrain){
        if(Keyboard.isKeyDown(Keyboard.KEY_UP)){
           this.currentSpeed= RUN_SPEED;
        }else if(Keyboard.isKeyDown(Keyboard.KEY_DOWN)){
            this.currentSpeed= -RUN_SPEED;
        }else {
            this.currentSpeed = 0;
        }
        if(Keyboard.isKeyDown(Keyboard.KEY_LEFT)){
            this.currentTurnSpeed= TURN_SPEED;
        } else if(Keyboard.isKeyDown(Keyboard.KEY_RIGHT)){
```

```
            this.currentTurnSpeed= -TURN_SPEED;
        }else {
            this.currentTurnSpeed = 0;
        }
        if(Keyboard.isKeyDown(Keyboard.KEY_J)){
            if(super.getPosition().y ==
terrain.getHeightOfTerrain(super.getPosition().x,super.getPosition().z)) {
                jump();
            }
        }

    }

    private void jump(){
        this.upwardsSpeed = JUMP_POWER;
    }

}
```

[11]

```
public class RawModel {

      private int vaoID;
      private int vertexCount;

      public RawModel(int vaoID, int vertexCount){
            this.vaoID = vaoID;
            this.vertexCount = vertexCount;
      }

      public int getVaoID() {
            return vaoID;
      }

      public int getVertexCount() {
            return vertexCount;
      }
```

```
}
public class TexturedModel {

	private RawModel rawModel;
	private ModelTexture texture;


	public TexturedModel(RawModel model, ModelTexture texture){
		this.rawModel = model;
		this.texture = texture;
	}

	public RawModel getRawModel() {
		return rawModel;
	}

	public ModelTexture getTexture() {
		return texture;
	}

}
ublic class OBJFileLoader {

    private static final String RES_LOC = "res/";

    public static ModelData loadOBJ(String objFileName) {
        FileReader isr = null;
        File objFile = new File(RES_LOC + objFileName + ".obj");
        try {
            isr = new FileReader(objFile);
        } catch (FileNotFoundException e) {
            System.err.println("File not found in res; don't use any extention");
```

```
        }
        BufferedReader reader = new BufferedReader(isr);
        String line;
        List<Vertex> vertices = new ArrayList<Vertex>();
        List<Vector2f> textures = new ArrayList<Vector2f>();
        List<Vector3f> normals = new ArrayList<Vector3f>();
        List<Integer> indices = new ArrayList<Integer>();



        try {
            while (true) {
                line = reader.readLine();
                if (line.startsWith("v ")) {
                    String[] currentLine = line.split(" ");
                    Vector3f vertex = new Vector3f((float)
Float.valueOf(currentLine[1]),
                            (float) Float.valueOf(currentLine[2]),
                            (float) Float.valueOf(currentLine[3]));

                    Vertex newVertex = new Vertex(vertices.size(), vertex);
                    vertices.add(newVertex);

                } else if (line.startsWith("vt ")) {
                    String[] currentLine = line.split(" ");
                    Vector2f texture = new Vector2f((float)
Float.valueOf(currentLine[1]),
                            (float) Float.valueOf(currentLine[2]));
                    textures.add(texture);
                } else if (line.startsWith("vn ")) {
                    String[] currentLine = line.split(" ");
                    Vector3f normal = new Vector3f((float)
Float.valueOf(currentLine[1]),
                            (float) Float.valueOf(currentLine[2]),
                            (float) Float.valueOf(currentLine[3]));
                    normals.add(normal);
                } else if (line.startsWith("f ")) {
```

```
                    break;
                }
            }
            while (line != null && line.startsWith("f ")) {
                String[] currentLine = line.split(" ");
                String[] vertex1 = currentLine[1].split("/");
                String[] vertex2 = currentLine[2].split("/");
                String[] vertex3 = currentLine[3].split("/");
                processVertex(vertex1, vertices, indices);
                processVertex(vertex2, vertices, indices);
                processVertex(vertex3, vertices, indices);
                line = reader.readLine();
            }
            reader.close();
        } catch (IOException e) {
            System.err.println("Error reading the file");
        }
        removeUnusedVertices(vertices);
        float[] verticesArray = new float[vertices.size() * 3];
        float[] texturesArray = new float[vertices.size() * 2];
        float[] normalsArray = new float[vertices.size() * 3];
        float furthest = convertDataToArrays(vertices, textures, normals,
verticesArray,
                texturesArray, normalsArray);
        int[] indicesArray = convertIndicesListToArray(indices);
        ModelData data = new ModelData(verticesArray, texturesArray, normalsArray,
indicesArray,
                furthest);
        return data;
    }


    private static void processVertex(String[] vertex, List<Vertex> vertices,
List<Integer> indices) {
        int index = Integer.parseInt(vertex[0]) - 1;
        Vertex currentVertex = vertices.get(index);
        int textureIndex = Integer.parseInt(vertex[1]) - 1;
        int normalIndex = Integer.parseInt(vertex[2]) - 1;
```

```
        if (!currentVertex.isSet()) {
            currentVertex.setTextureIndex(textureIndex);
            currentVertex.setNormalIndex(normalIndex);
            indices.add(index);
        } else {
            dealWithAlreadyProcessedVertex(currentVertex, textureIndex,
normalIndex, indices,
                    vertices);
        }
    }

    private static int[] convertIndicesListToArray(List<Integer> indices) {
        int[] indicesArray = new int[indices.size()];
        for (int i = 0; i < indicesArray.length; i++) {
            indicesArray[i] = indices.get(i);
        }
        return indicesArray;
    }

    private static float convertDataToArrays(List<Vertex> vertices, List<Vector2f>
textures,
                                             List<Vector3f> normals, float[]
verticesArray, float[] texturesArray,
                                             float[] normalsArray) {
        float furthestPoint = 0;
        for (int i = 0; i < vertices.size(); i++) {
            Vertex currentVertex = vertices.get(i);
            if (currentVertex.getLength() > furthestPoint) {
                furthestPoint = currentVertex.getLength();
            }
            Vector3f position = currentVertex.getPosition();
            Vector2f textureCoord = textures.get(currentVertex.getTextureIndex());
            Vector3f normalVector = normals.get(currentVertex.getNormalIndex());
            verticesArray[i * 3] = position.x;
            verticesArray[i * 3 + 1] = position.y;
            verticesArray[i * 3 + 2] = position.z;
            texturesArray[i * 2] = textureCoord.x;
```

```
            texturesArray[i * 2 + 1] = 1 - textureCoord.y;
            normalsArray[i * 3] = normalVector.x;
            normalsArray[i * 3 + 1] = normalVector.y;
            normalsArray[i * 3 + 2] = normalVector.z;
        }
        return furthestPoint;
    }

    private static void dealWithAlreadyProcessedVertex(Vertex previousVertex, int
newTextureIndex,
                                                       int newNormalIndex,
List<Integer> indices, List<Vertex> vertices) {
        if (previousVertex.hasSameTextureAndNormal(newTextureIndex,
newNormalIndex)) {
            indices.add(previousVertex.getIndex());
        } else {
            Vertex anotherVertex = previousVertex.getDuplicateVertex();
            if (anotherVertex != null) {
                dealWithAlreadyProcessedVertex(anotherVertex, newTextureIndex,
newNormalIndex,
                        indices, vertices);
            } else {
                Vertex duplicateVertex = new Vertex(vertices.size(),
previousVertex.getPosition());
                duplicateVertex.setTextureIndex(newTextureIndex);
                duplicateVertex.setNormalIndex(newNormalIndex);
                previousVertex.setDuplicateVertex(duplicateVertex);
                vertices.add(duplicateVertex);
                indices.add(duplicateVertex.getIndex());
            }

        }
    }

    private static void removeUnusedVertices(List<Vertex> vertices){
        for(Vertex vertex:vertices){
            if(!vertex.isSet()){
```

```
                vertex.setTextureIndex(0);
                vertex.setNormalIndex(0);
            }
        }
    }

}
public class DisplayManager {

      private static final int WIDTH = 1280;
      private static final int HEIGHT = 1024;
      private static final int FPS_CAP = 145;

      private static long lastFrameTime;
      private static float delta;

      public static void createDisplay(){
            ContextAttribs attribs = new ContextAttribs(3,2)
            .withForwardCompatible(true)
            .withProfileCore(true);

            try {
                  Display.setDisplayMode(new DisplayMode(WIDTH,HEIGHT));
                  Display.create(new PixelFormat(), attribs);
                  Display.setTitle("Test Engine");
            } catch (LWJGLException e) {
                  e.printStackTrace();
            }

            GL11.glViewport(0,0, WIDTH, HEIGHT);
            lastFrameTime = getCurrentTime();

      }

      public static void updateDisplay(){
```

```
		Display.sync(FPS_CAP);
		Display.update();

		long currentFrameTime = getCurrentTime();
		delta  = (currentFrameTime - lastFrameTime)/1000f;
		lastFrameTime = currentFrameTime;
	}

	public static float getFrameTimeSeconds(){
		return delta;
	}

	public static void closeDisplay(){

		Display.destroy();

	}

	private static long getCurrentTime(){
		return Sys.getTime()*1000/Sys.getTimerResolution();
	}
}
public class EntityRenderer {

	private StaticShader shader;

	public EntityRenderer(StaticShader shader,Matrix4f projectionMatrix) {
		this.shader = shader;
		shader.start();
		shader.loadProjectionMatrix(projectionMatrix);
		shader.stop();
	}

	public void render(Map<TexturedModel, List<Entity>> entities) {
```

```
		for (TexturedModel model : entities.keySet()) {
			prepareTexturedModel(model);
			List<Entity> batch = entities.get(model);
			for (Entity entity : batch) {
				prepareInstance(entity);
				GL11.glDrawElements(GL11.GL_TRIANGLES,
model.getRawModel().getVertexCount(),
					GL11.GL_UNSIGNED_INT, 0);
			}
			unbindTexturedModel();
		}
	}

	private void prepareTexturedModel(TexturedModel model) {
		RawModel rawModel = model.getRawModel();
		GL30.glBindVertexArray(rawModel.getVaoID());
		GL20.glEnableVertexAttribArray(0);
		GL20.glEnableVertexAttribArray(1);
		GL20.glEnableVertexAttribArray(2);
		ModelTexture texture = model.getTexture();
		if(texture.isHasTransparency()){
			MasterRenderer.disableCulling();
		}
		shader.loadFakeLightingVariable(texture.isUseFakeLighting());
		shader.loadShineVariables(texture.getShineDamper(),
texture.getReflectivity());
		GL13.glActiveTexture(GL13.GL_TEXTURE0);
		GL11.glBindTexture(GL11.GL_TEXTURE_2D, model.getTexture().getID());
	}

	private void unbindTexturedModel() {
		MasterRenderer.enableCulling();
		GL20.glDisableVertexAttribArray(0);
		GL20.glDisableVertexAttribArray(1);
		GL20.glDisableVertexAttribArray(2);
		GL30.glBindVertexArray(0);
```

```
	}


	private void prepareInstance(Entity entity) {
		Matrix4f transformationMatrix =
Maths.createTransformationMatrix(entity.getPosition(),
				entity.getRotX(), entity.getRotY(), entity.getRotZ(),
entity.getScale());
		shader.loadTransformationMatrix(transformationMatrix);
	}


}
public class Loader {


	private List<Integer> vaos = new ArrayList<Integer>();
	private List<Integer> vbos = new ArrayList<Integer>();
	private List<Integer> textures = new ArrayList<Integer>();


	public RawModel loadToVAO(float[] positions,float[] textureCoords,float[]
normals,int[] indices){
		int vaoID = createVAO();
		bindIndicesBuffer(indices);
		storeDataInAttributeList(0,3,positions);
		storeDataInAttributeList(1,2,textureCoords);
		storeDataInAttributeList(2,3,normals);
		unbindVAO();
		return new RawModel(vaoID,indices.length);
	}


	public int loadTexture(String fileName) {
		Texture texture = null;
		try {
			texture = TextureLoader.getTexture("PNG",
					new FileInputStream("res/" + fileName + ".png"));
		} catch (Exception e) {
			e.printStackTrace();
			System.err.println("Tried to load texture " + fileName + ".png ,
didn't work");
```

```
                System.exit(-1);
            }
            textures.add(texture.getTextureID());
            return texture.getTextureID();
      }

      public void cleanUp(){
            for(int vao:vaos){
                GL30.glDeleteVertexArrays(vao);
            }
            for(int vbo:vbos){
                GL15.glDeleteBuffers(vbo);
            }
            for(int texture:textures){
                GL11.glDeleteTextures(texture);
            }
      }

      private int createVAO(){
            int vaoID = GL30.glGenVertexArrays();
            vaos.add(vaoID);
            GL30.glBindVertexArray(vaoID);
            return vaoID;
      }

      private void storeDataInAttributeList(int attributeNumber, int
coordinateSize,float[] data){
            int vboID = GL15.glGenBuffers();
            vbos.add(vboID);
            GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vboID);
            FloatBuffer buffer = storeDataInFloatBuffer(data);
            GL15.glBufferData(GL15.GL_ARRAY_BUFFER, buffer, GL15.GL_STATIC_DRAW);

      GL20.glVertexAttribPointer(attributeNumber,coordinateSize,GL11.GL_FLOAT,fals
e,0,0);
            GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, 0);
      }
```

```
	private void unbindVAO(){
		GL30.glBindVertexArray(0);
	}

	private void bindIndicesBuffer(int[] indices){
		int vboID = GL15.glGenBuffers();
		vbos.add(vboID);
		GL15.glBindBuffer(GL15.GL_ELEMENT_ARRAY_BUFFER, vboID);
		IntBuffer buffer = storeDataInIntBuffer(indices);
		GL15.glBufferData(GL15.GL_ELEMENT_ARRAY_BUFFER, buffer,
GL15.GL_STATIC_DRAW);
	}

	private IntBuffer storeDataInIntBuffer(int[] data){
		IntBuffer buffer = BufferUtils.createIntBuffer(data.length);
		buffer.put(data);
		buffer.flip();
		return buffer;
	}

	private FloatBuffer storeDataInFloatBuffer(float[] data){
		FloatBuffer buffer = BufferUtils.createFloatBuffer(data.length);
		buffer.put(data);
		buffer.flip();
		return buffer;
	}




}
public class MasterRenderer {

	private static final float FOV = 70;
	private static final float NEAR_PLANE = 0.1f;
	private static final float FAR_PLANE = 10000;
```

```
	private static final float RED = 0.49f;
	private static final float GREEN = 0.89f;
	private static final float BLUE =0.98f;

	private Matrix4f projectionMatrix;

	private StaticShader shader = new StaticShader();
	private EntityRenderer renderer;

	private TerrainRenderer terrainRenderer;
	private TerrainShader terrainShader = new TerrainShader();


	private Map<TexturedModel,List<Entity>> entities = new
HashMap<TexturedModel,List<Entity>>();
	private List<Terrain> terrains = new ArrayList<Terrain>();

	public MasterRenderer(){
		enableCulling();
		createProjectionMatrix();
		renderer = new EntityRenderer(shader,projectionMatrix);
		terrainRenderer = new TerrainRenderer(terrainShader,projectionMatrix);
	}

	public static void enableCulling(){
		GL11.glEnable(GL11.GL_CULL_FACE);
		GL11.glCullFace(GL11.GL_BACK);
	}
	public static void disableCulling(){
		GL11.glDisable(GL11.GL_CULL_FACE);

	}

	public void render(Light sun,Camera camera){
		prepare();
```

```
		shader.start();
		shader.loadSkyColour(RED,GREEN,BLUE);
		shader.loadLight(sun);
		shader.loadViewMatrix(camera);
		renderer.render(entities);
		shader.stop();
		terrainShader.start();
		terrainShader.loadSkyColour(RED,GREEN,BLUE);
		terrainShader.loadLight(sun);
		terrainShader.loadViewMatrix(camera);
		terrainRenderer.render(terrains);
		terrainShader.stop();
		terrains.clear();
		entities.clear();
	}

	public void processTerrain(Terrain terrain){
		terrains.add(terrain);
	}

	public void processEntity(Entity entity){
		TexturedModel entityModel = entity.getModel();
		List<Entity> batch = entities.get(entityModel);
		if(batch!=null){
			batch.add(entity);
		}else{
			List<Entity> newBatch = new ArrayList<Entity>();
			newBatch.add(entity);
			entities.put(entityModel, newBatch);
		}
	}

	public void cleanUp(){
		shader.cleanUp();
		terrainShader.cleanUp();
```

```
	}

	public void prepare() {
		GL11.glEnable(GL11.GL_DEPTH_TEST);
		GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
		GL11.glClearColor(RED, GREEN, BLUE, 1);
	}

	private void createProjectionMatrix() {
		float aspectRatio = (float) Display.getWidth() / (float) Display.getHeight();
		float y_scale = (float) ((1f / Math.tan(Math.toRadians(FOV / 2f))) * aspectRatio);
		float x_scale = y_scale / aspectRatio;
		float frustum_length = FAR_PLANE - NEAR_PLANE;

		projectionMatrix = new Matrix4f();
		projectionMatrix.m00 = x_scale;
		projectionMatrix.m11 = y_scale;
		projectionMatrix.m22 = -((FAR_PLANE + NEAR_PLANE) / frustum_length);
		projectionMatrix.m23 = -1;
		projectionMatrix.m32 = -((2 * NEAR_PLANE * FAR_PLANE) / frustum_length);
		projectionMatrix.m33 = 0;
	}


}
public abstract class ShaderProgram {

	private int programID;
	private int vertexShaderID;
	private int fragmentShaderID;

	private static FloatBuffer matrixBuffer = BufferUtils.createFloatBuffer(16);

	public ShaderProgram(String vertexFile,String fragmentFile){
```

```
		vertexShaderID = loadShader(vertexFile,GL20.GL_VERTEX_SHADER);
		fragmentShaderID = loadShader(fragmentFile,GL20.GL_FRAGMENT_SHADER);
		programID = GL20.glCreateProgram();
		GL20.glAttachShader(programID, vertexShaderID);
		GL20.glAttachShader(programID, fragmentShaderID);
		bindAttributes();
		GL20.glLinkProgram(programID);
		GL20.glValidateProgram(programID);
		getAllUniformLocations();
	}

	protected abstract void getAllUniformLocations();

	protected int getUniformLocation(String uniformName){
		return GL20.glGetUniformLocation(programID,uniformName);
	}

	public void start(){
		GL20.glUseProgram(programID);
	}

	public void stop(){
		GL20.glUseProgram(0);
	}

	public void cleanUp(){
		stop();
		GL20.glDetachShader(programID, vertexShaderID);
		GL20.glDetachShader(programID, fragmentShaderID);
		GL20.glDeleteShader(vertexShaderID);
		GL20.glDeleteShader(fragmentShaderID);
		GL20.glDeleteProgram(programID);
	}

	protected abstract void bindAttributes();
```

```
	protected void bindAttribute(int attribute, String variableName){
		GL20.glBindAttribLocation(programID, attribute, variableName);
	}

	protected void loadFloat(int location, float value){
		GL20.glUniform1f(location, value);
	}

	protected void loadInt(int location, int value){
		GL20.glUniform1i(location, value);
	}

	protected void loadVector(int location, Vector3f vector){
		GL20.glUniform3f(location,vector.x,vector.y,vector.z);
	}

	protected void loadBoolean(int location, boolean value){
		float toLoad = 0;
		if(value){
			toLoad = 1;
		}
		GL20.glUniform1f(location, toLoad);
	}

	protected void loadMatrix(int location, Matrix4f matrix){
		matrix.store(matrixBuffer);
		matrixBuffer.flip();
		GL20.glUniformMatrix4(location, false, matrixBuffer);
	}

	private static int loadShader(String file, int type){
		StringBuilder shaderSource = new StringBuilder();
		try{
			BufferedReader reader = new BufferedReader(new
FileReader(file));
```

```
                String line;
                while((line = reader.readLine())!=null){
                      shaderSource.append(line).append("//\n");
                }
                reader.close();
            }catch(IOException e){
                e.printStackTrace();
                System.exit(-1);
            }
            int shaderID = GL20.glCreateShader(type);
            GL20.glShaderSource(shaderID, shaderSource);
            GL20.glCompileShader(shaderID);
            if(GL20.glGetShaderi(shaderID, GL20.GL_COMPILE_STATUS )==
GL11.GL_FALSE){
                System.out.println(GL20.glGetShaderInfoLog(shaderID, 500));
                System.err.println("Could not compile shader!");
                System.exit(-1);
            }
            return shaderID;
      }


}
public class TerrainShader extends ShaderProgram{



      private static final String VERTEX_FILE =
"src/shaders/terrainVertexShader.txt";
      private static final String FRAGMENT_FILE =
"src/shaders/terrainFragmentShader.txt";


      private int location_transformationMatrix;
      private int location_projectionMatrix;
      private int location_viewMatrix;
      private int location_lightPosition;
      private int location_lightColour;
      private int location_shineDamper;
      private int location_reflectivity;
```

```
	private int location_skyColour;
	private int location_backgroundTexture;
	private int location_rTexture;
	private int location_gTexture;
	private int location_bTexture;
	private int location_blendMap;

	public TerrainShader() {
		super(VERTEX_FILE, FRAGMENT_FILE);
	}

	@Override
	protected void bindAttributes() {
		super.bindAttribute(0, "position");
		super.bindAttribute(1, "textureCoordinates");
		super.bindAttribute(2, "normal");
	}

	@Override
	protected void getAllUniformLocations() {
		location_transformationMatrix =
super.getUniformLocation("transformationMatrix");
		location_projectionMatrix =
super.getUniformLocation("projectionMatrix");
		location_viewMatrix = super.getUniformLocation("viewMatrix");
		location_lightPosition = super.getUniformLocation("lightPosition");
		location_lightColour = super.getUniformLocation("lightColour");
		location_shineDamper = super.getUniformLocation("shineDamper");
		location_reflectivity = super.getUniformLocation("reflectivity");
		location_skyColour = super.getUniformLocation("skyColour");
		location_backgroundTexture =
super.getUniformLocation("backgroundTexture");
		location_rTexture = super.getUniformLocation("rTexture");
		location_gTexture = super.getUniformLocation("gTexture");
		location_bTexture = super.getUniformLocation("bTexture");
		location_blendMap = super.getUniformLocation("blendMap");
```

```
}

public void connectTextureUnits(){
	super.loadInt(location_backgroundTexture, 0);
	super.loadInt(location_rTexture, 1);
	super.loadInt(location_gTexture, 2);
	super.loadInt(location_bTexture, 3);
	super.loadInt(location_blendMap, 4);

}

public void loadSkyColour(float r,float g,float b){
	super.loadVector(location_skyColour,new Vector3f(r,g,b));
}


public void loadShineVariables(float damper,float reflectivity){
	super.loadFloat(location_shineDamper, damper);
	super.loadFloat(location_reflectivity, reflectivity);
}

public void loadTransformationMatrix(Matrix4f matrix){
	super.loadMatrix(location_transformationMatrix, matrix);
}

public void loadLight(Light light){
	super.loadVector(location_lightPosition, light.getPosition());
	super.loadVector(location_lightColour, light.getColour());
}

public void loadViewMatrix(Camera camera){
	Matrix4f viewMatrix = Maths.createViewMatrix(camera);
	super.loadMatrix(location_viewMatrix, viewMatrix);
}
```

```
	public void loadProjectionMatrix(Matrix4f projection){
		super.loadMatrix(location_projectionMatrix, projection);
	}



}
public class Terrain {

	private static final float SIZE = 2000;
	private static final float MAX_HEIGHT = 25;
	private static final float MAX_PIXEL_COLOUR = 256*256*256;
	//private static final int VERTEX_COUNT = 128;




	private float x;
	private float z;
	private RawModel model;
	private TerrainTexturePack texturePack;
	private TerrainTexture blendMap;

	private float[][] heights;

	public Terrain(int gridX, int gridZ, Loader loader, TerrainTexturePack
texturePack,TerrainTexture blendMap, String heightMap){
		this.texturePack = texturePack;
		this.blendMap = blendMap;
		this.x = gridX * SIZE + 1000;
		this.z = gridZ * SIZE + 1000;
		this.model = generateTerrain(loader, heightMap);
	}




	public float getX() {
		return x;
```

```
	}



	public float getZ() {
		return z;
	}



	public RawModel getModel() {
		return model;
	}

	public TerrainTexturePack getTexturePack() {
		return texturePack;
	}

	public TerrainTexture getBlendMap() {
		return blendMap;
	}

	public float getHeightOfTerrain(float worldX, float worldZ){
		float terrainX = worldX - this.x;
		float terrainZ = worldZ - this.z;
		float gridSquareSize = SIZE / ((float) heights.length - 1);
		int gridX = (int)Math.floor(terrainX/gridSquareSize);
		int gridZ = (int)Math.floor(terrainZ/gridSquareSize);
		if(gridX >= heights.length-1 ||gridZ >= heights.length-1 || gridX <
0|| gridZ < 0){
			return 0;
		}
		float xCoord = (terrainX % gridSquareSize)/ gridSquareSize;
		float zCoord = (terrainZ % gridSquareSize)/ gridSquareSize;
		float answer;
		if (xCoord <= (1-zCoord)) {
```

```
                  answer = Maths.barryCentric(new Vector3f(0,
heights[gridX][gridZ], 0), new Vector3f(1,
                                          heights[gridX + 1][gridZ], 0), new
Vector3f(0,
                                          heights[gridX][gridZ + 1], 1), new
Vector2f(xCoord, zCoord));
            } else {
                  answer = Maths.barryCentric(new Vector3f(1, heights[gridX +
1][gridZ], 0), new Vector3f(1,
                                          heights[gridX + 1][gridZ + 1], 1), new
Vector3f(0,
                                          heights[gridX][gridZ + 1], 1), new
Vector2f(xCoord, zCoord));
            }
            return answer;
      }

      private RawModel generateTerrain(Loader loader,String heightMap){
            BufferedImage image = null;
            try {
                  image = ImageIO.read(new File("res/"+heightMap +".png") );
            } catch (IOException e) {
                  e.printStackTrace();
            }

            int VERTEX_COUNT = image.getHeight();

            heights = new float[VERTEX_COUNT][VERTEX_COUNT];

            int count = VERTEX_COUNT * VERTEX_COUNT;
            float[] vertices = new float[count * 3];
            float[] normals = new float[count * 3];
            float[] textureCoords = new float[count*2];
            int[] indices = new int[6*(VERTEX_COUNT-1)*(VERTEX_COUNT-1)];
            int vertexPointer = 0;
            for(int i=0;i<VERTEX_COUNT;i++){
                  for(int j=0;j<VERTEX_COUNT;j++){
```

```
                        vertices[vertexPointer*3] = (float)j/((float)VERTEX_COUNT -
1) * SIZE;
                        float height  = getHeight(j,i, image);
                        heights[j][i] = height;
                        vertices[vertexPointer*3+1] = height;
                        vertices[vertexPointer*3+2] = (float)i/((float)VERTEX_COUNT
- 1) * SIZE;
                        Vector3f normal = calculateNormal(j, i, image);
                        normals[vertexPointer*3] = normal.x;
                        normals[vertexPointer*3+1] = normal.y;
                        normals[vertexPointer*3+2] = normal.z;
                        textureCoords[vertexPointer*2] =
(float)j/((float)VERTEX_COUNT - 1);
                        textureCoords[vertexPointer*2+1] =
(float)i/((float)VERTEX_COUNT - 1);
                        vertexPointer++;
                  }
            }
            int pointer = 0;
            for(int gz=0;gz<VERTEX_COUNT-1;gz++){
                  for(int gx=0;gx<VERTEX_COUNT-1;gx++){
                        int topLeft = (gz*VERTEX_COUNT)+gx;
                        int topRight = topLeft + 1;
                        int bottomLeft = ((gz+1)*VERTEX_COUNT)+gx;
                        int bottomRight = bottomLeft + 1;
                        indices[pointer++] = topLeft;
                        indices[pointer++] = bottomLeft;
                        indices[pointer++] = topRight;
                        indices[pointer++] = topRight;
                        indices[pointer++] = bottomLeft;
                        indices[pointer++] = bottomRight;
                  }
            }
            return loader.loadToVAO(vertices, textureCoords, normals, indices);
      }

      private Vector3f calculateNormal(int x, int z, BufferedImage image){
```

```
            float heightL = getHeight(x-1,z,image);
            float heightR = getHeight(x+1,z,image);
            float heightD = getHeight(x,z-1,image);
            float heightU = getHeight(x,z+1,image);
            Vector3f normal = new Vector3f( heightL - heightR, 2f, heightD -
heightU);
            normal.normalise();
            return normal;

      }

      private float getHeight(int x, int z, BufferedImage image){
            if(x<0 || x>= image.getHeight() || z<0 || z>= image.getHeight()){
                  return 0;
            }
            float height = image.getRGB(x,z);
            height += MAX_PIXEL_COLOUR/2f;
            height /=MAX_PIXEL_COLOUR/2f;
            height *= MAX_HEIGHT;
            return height;
      }

}
public class ModelTexture {

      private int textureID;

      private float shineDamper = 1;
      private float reflectivity = 0;

      private boolean hasTransparency = false;
      private boolean useFakeLighting = false;

      public ModelTexture(int texture){
            this.textureID = texture;
      }
```

```
public int getID(){
	return textureID;
}

public float getShineDamper() {
	return shineDamper;
}

public void setShineDamper(float shineDamper) {
	this.shineDamper = shineDamper;
}

public float getReflectivity() {
	return reflectivity;
}

public void setReflectivity(float reflectivity) {
	this.reflectivity = reflectivity;
}


public boolean isHasTransparency() {
	return hasTransparency;
}

public void setHasTransparency(boolean hasTransparency) {
	this.hasTransparency = hasTransparency;
}

public boolean isUseFakeLighting() {
	return useFakeLighting;
}

public void setUseFakeLighting(boolean useFakeLighting) {
```

```
            this.useFakeLighting = useFakeLighting;
      }
}
public class Maths {


      public static float barryCentric(Vector3f p1, Vector3f p2, Vector3f p3,
Vector2f pos) {
            float det = (p2.z - p3.z) * (p1.x - p3.x) + (p3.x - p2.x) * (p1.z -
p3.z);
            float l1 = ((p2.z - p3.z) * (pos.x - p3.x) + (p3.x - p2.x) * (pos.y -
p3.z)) / det;
            float l2 = ((p3.z - p1.z) * (pos.x - p3.x) + (p1.x - p3.x) * (pos.y -
p3.z)) / det;
            float l3 = 1.0f - l1 - l2;
            return l1 * p1.y + l2 * p2.y + l3 * p3.y;
      }


      public static Matrix4f createTransformationMatrix(Vector3f translation,
float rx, float ry,
                  float rz, float scale) {
            Matrix4f matrix = new Matrix4f();
            matrix.setIdentity();
            Matrix4f.translate(translation, matrix, matrix);
            Matrix4f.rotate((float) Math.toRadians(rx), new Vector3f(1,0,0),
matrix, matrix);
            Matrix4f.rotate((float) Math.toRadians(ry), new Vector3f(0,1,0),
matrix, matrix);
            Matrix4f.rotate((float) Math.toRadians(rz), new Vector3f(0,0,1),
matrix, matrix);
            Matrix4f.scale(new Vector3f(scale,scale,scale), matrix, matrix);
            return matrix;
      }


      public static Matrix4f createViewMatrix(Camera camera) {
            Matrix4f viewMatrix = new Matrix4f();
            viewMatrix.setIdentity();
            Matrix4f.rotate((float) Math.toRadians(camera.getPitch()), new
Vector3f(1, 0, 0), viewMatrix,
                        viewMatrix);
```

```
            Matrix4f.rotate((float) Math.toRadians(camera.getYaw()), new
Vector3f(0, 1, 0), viewMatrix, viewMatrix);

            Vector3f cameraPos = camera.getPosition();

            Vector3f negativeCameraPos = new Vector3f(-cameraPos.x,-cameraPos.y,-
cameraPos.z);

            Matrix4f.translate(negativeCameraPos, viewMatrix, viewMatrix);

            return viewMatrix;

      }


}
```